COMPARISON OF SEARCH ALGORITHMS IN TWO-STAGE NEURAL NETWORK TRAINING FOR OPTICAL CHARACTER RECOGNITION OF HANDWRITTEN DIGITS

A thesis presented to the faculty of the Graduate School of Western Carolina University in partial fulfillment of the requirements for the degree of Master of Science in Technology.

By

Patrik Wayne Gilley

Director: Dr. Yanjun Yan
Associate Professor
School of Engineering and Technology

Committee Members: Dr. Paul Yanik, School of Engineering and Technology
Dr. Bora Karayaka, School of Engineering and Technology
Dr. Peter Tay, School of Engineering and Technology

April 2020

This work is dedicated to:

My parents for their unending support and love.

Joshua Turnbull, for being my intellectual sparring partner and closest friend.

My sister, for inspiring me to chase my creativity wherever it takes me.

ACKNOWLEDGEMENTS

I would like to acknowledge my thesis advisor Dr. Yanjun Yan for her constant guidance and assistance. I will never forget her dedication to working with me on my thesis in the face of great personal loss. I wish to express my gratitude to her for all of the invaluable help she gave me in publishing my first paper.

I also wish to thank Dr. Paul Yanik for all of the help he has given me during my time at Western Carolina University. Convincing me to attend graduate school has helped me grow in ways that I could not have imagined, and the advice that you have given me both academic and professional has helped me more times than I can remember.

I would like to acknowledge my parents for constantly supporting in life no matter how large or small my problems are. The time you spent fostering my curiosity and challenging me to grow beyond my limits has taught me some very valuable lessons. I can honestly say that I would not be where I am today without you.

Finally, I would like to acknowledge the other graduate students I worked with while completing my degree for their constant companionship and advice. In particular, I would like to acknowledge Jeremy Howell, Connor McIntyre, Eric Meyers, Prem Kumar, Anik Tahabilder, and especially my longtime friend Joshua Turnbull. They made these past few years much easier and more enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

COMPARISON OF SEARCH ALGORITHMS IN TWO-STAGE NEURAL NETWORK TRAINING FOR OPTICAL CHARACTER RECOGNITION OF HANDWRITTEN DIGITS

Patrik Wayne Gilley, M.S.T.

Western Carolina University (April 2020)

Director: Dr. Yanjun Yan

Neural networks are a powerful machine learning technique that give a system the ability to develop multiple levels of abstraction to interpret data. Such networks require training to develop the neural layers and neuron weights in order to form reasonable conclusions from the data being interpreted. The conventional training method is to use backpropagation to feed the error between a neural network's actual output and desired output back through the neural network to adjust the neural synapses' weights to minimize such errors on subsequent training data. However, backpropagation has several limitations to its effectiveness in training neural networks. The most relevant limitation of backpropagation is that it tends to become trapped in local optimum solutions. This research studied the effectiveness in using search algorithms to improve upon a feed-forward Multi-Layer Perceptron artificial neural network trained by backpropagation in classifying handwritten digits. The search algorithms used for this research were the Bare Bones Fireworks Algorithm (BBFWA), Canonical Particle Swarm Optimization (PSO), and Cooperative Particle Swarm Optimization (CPSO) algorithms. Two sets of parameters for the BBFWA were tested in this study in order to examine the effects of parameters on the algorithm. The handwritten digit classification data was carried out on the MNIST handwritten character database, a common benchmark for handwritten character recognition. A neural network was trained with backpropagation, and then the search algorithms were seeded with its weights so that they could search for better

neural network weight configurations. The complexity of using images of handwritten characters with a feed-forward Multi-Layer Perceptron resulted in a high degree of dimensionality in the problem, which negatively impacted the Particle Swarm Optimization algorithms. To analyze the impact of the problem dimensionality, the neural network was also tested with a PCA compressed MNIST database. It was found that the BBFWA performed the best out of the three algorithms on both datasets, as it was able to consistently improve upon the performance of the original neural networks. Between the two sets of BBFWA parameters, the simulation results indicated that the second parameter set outperformed the first parameter set in terms of both classification accuracy and fitness trends.

CHAPTER 1: INTRODUCTION

1.1 Background

Optical Character Recognition (OCR) is a specialization of image processing that concerns character reading and recognition by computers. Artificial neural networks (ANN) are one of the most commonly used techniques in implementing OCR systems, as they can simplify systems and maintain the ability to perform well on unseen character sets [1]. An ANN is modelled after the structure and behavior of neurons in the human brain, and is capable of learning complex patterns from data sets to perform tasks such as classifying images of handwritten digits. As such, an ANN is capable of learning a large amount of complex non-linear equations within an organized framework that can theoretically learn any behavior desired of it by its designers [2].

Artificial neural networks learn using a training algorithm, which encourages the ANN to match a desired behavior. The conventional training algorithm for most ANN applications is the Backpropagation (BP) algorithm [2]. There are several algorithms that fall under the category of BP, but they all share a few common traits. The BP algorithm is a gradient descent algorithm that can identify what neurons of an ANN are most responsible for wrong behavior during a single evaluation of all training data, or epoch, of the training process. By adjusting the weights of the ANN neurons based on how responsible each neuron is for the wrong behavior, a BP algorithm produces ordered, steady improvement in ANN performance on a task. This encourages the ANN to extract fundamental patterns about the data that it trains on. In turn, the model the ANN learns can better generalize to unseen data samples. However, the BP algorithm has several drawbacks to it. Due to the gradient descent nature of BP algorithms, the training process can get stuck in local optimal solutions. Additionally, as the data samples the ANN trains on increase in complexity, the computational costs of running modern BP algorithms can significantly increase.

Search algorithms are evolutionary algorithms that use meta-heuristic rules to search a solution space consisting of all possible solutions to an optimization problem [3]. These algorithms have been successfully applied to training ANN across a variety of applications, such as camera OCR of vehicle VIN numbers, medical data mining and diagnosis, and handwritten digit OCR [1,4–7]. This research utilizes several search algorithms: the Particle Swarm Optimization (PSO) algorithm [8], the Cooperative Particle Swarm Optimization (CPSO) algorithm [4], and the Bare Bones Fireworks Algorithm (BBFWA) [9]. The PSO algorithm has been successfully applied in ANN training since it was proposed by Eberhart and Kennedy in 1995 [8]. The PSO algorithm uses a swarm of particles to "fly" around a solution space, allowing it to test multiple variations of an ANN nodal weight configuration simultaneously. This allows for a more robust search of ANN configurations that allows the PSO algorithm to see possibilities outside of the current training path. The Fireworks Algorithm is modelled after firework explosions. It searches a solution space by setting off fireworks and generating sparks within the explosion range of each firework. By evaluating each spark as a solution and altering both the position and explosion radius of fireworks, the Fireworks Algorithm can efficiently explore a solution space and find an effective global optimum solution [9].

The heuristic nature of search algorithms minimizes the risk of getting trapped in local optima. This gives search algorithms an advantage over BP, as they can explore solutions that BP algorithms typically would not. However, PSO and PSO-based algorithms can struggle to converge to an effective ANN configuration in problems with high dimensionality such as image classification in feedforward Multi-Layer Perceptron (MLP) networks [7].

1.2 Study Objectives

This study explores the applicability of search algorithms to the problem of ANN training for pattern recognition applications. Pattern recognition applications vary widely in terms of both data pattern and source data types, from medical data with a small number of discrete features per data sample [5] to image classification with many features representing image pixels [7]. The dimensionality of pattern recognition data samples is an important consideration in any ANN design, and it has a significant impact on the performance of search algorithms such as the PSO search algorithm [4].

The purpose of the study is to compare the properties of the BBFWA, PSO, and CPSO algorithms in ANN pattern recognition applications with a huge dimensionality. The pattern recognition application that will be used as a test bed for this paper is handwritten OCR, as handwritten OCR provides a sufficiently large dimensionality to challenge the search algorithms. In order to provide a common ground for the comparison, this paper will employ a two-stage ANN training system with BP and search optimization algorithms. The two-stage training system will first train a feedforward MLP to classify the MNIST handwritten character database using conjugate-gradient BP. The BBFWA, PSO, and CPSO search algorithms will then be applied to improve upon the BP trained MLP. This two-stage training system uses transfer learning to allow the search algorithms to utilize what the BP trained MLP ANN learned. According to Pan and Yang, transfer learning is a process of transfering knowledge from a machine learning model to a different model. This process is useful in reducing the expense of collecting and labelling new data, or training an entirely new model from scratch. In ANN, this process can manifest as taking a very general ANN and adjusting it for a more specific application [10]. Utilizing transfer learning to start all of the search algorithms at the same pre-trained ANN allows for a common baseline to compare the performance of each algorithm against the rest of them. The algorithms will be judged on classification accuracy and the overall fitness that they achieve.

CHAPTER 2: LITERATURE SURVEY

2.1 Artificial Neural Networks

Artificial neural networks are a machine learning technique that mimics how a human brain processes information. Negnevitsky gives a general overview of ANN in his book on artificial intelligence [2]. Neurons in the human brain have a large amount of input signals, with each input signal having a different weight or importance to that neuron. Perceptrons, one of the basic nodal structures of ANN, are very similar to human neurons in how they operate. In theory a neural network has multiple layers of nodes, with every node in one layer connected to every node in the succeeding layer. At a minimum, a neural network has an input and output layer, with additional layers between the two being referred to as hidden layers. These hidden layers are optional design choices and can be added to the neural network at the discretion of the designer. Hidden layers allow for a greater range of nodal functions and features to be learned by a neural network, although the operations of the hidden layers are not necessarily apparent to humans [2].

Every layer in a neural network consists of a collection of nodes. Each individual node passes all the inputs from the previous network layer through an activation function to determine its output value. Each layer in an ANN applies the same activation function to all of the nodes within that layer, although each ANN layer can use a different activation function from what the other layers use. The only layer that does not use an activation function is the input layer, as the nodes in the input layer simply take on the value of the data sample given to them. Each input connection to a node has a weighted coefficient that is multiplied with the numerical input of the node, referred to as a nodal weight. Nodal weights in an ANN represent the importance of an input to a node's output. By varying the values of these nodal weights, it becomes possible to adjust the behavior of each node. This nodal weight adjustment process is the core problem of training an ANN, as this often requires adjusting thousands of nodal weights across several layers to achieve a desired behavior.

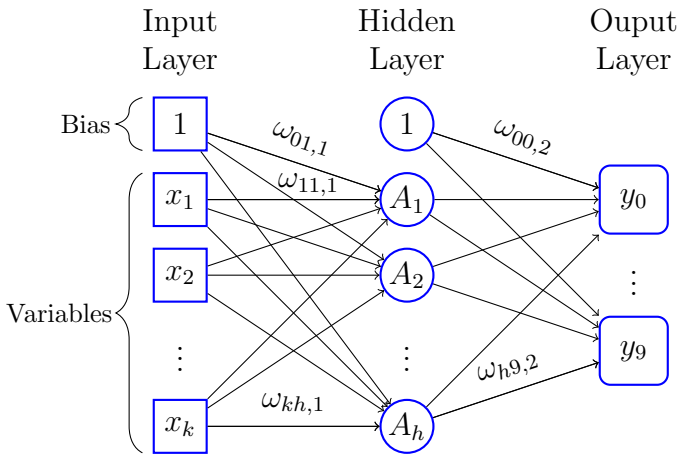

Figure 2.1: A Feedforward MLP Artificial Neural Network

The neural network architecture used in this study is the feedforward MLP architecture. An MLP network is composed of multiple layers of perceptron nodes, with information entering the input layer and sequentially proceeding to the output layer with no data feeding back to previous layers. Fig. 2.1 shows an example of the feedforward MLP neural network architecture. Each layer with the exception of the output layer has a bias node that constantly outputs a value of 1. The ω values represent individual nodal weights. These can be grouped together into nodal weight matrices that represent the entirety of the connections between two layers.

Because of the adaptability and ability to model complex non-linear relationships, ANN are applicable in a wide variety of fields. Shah et al. used an ANN to implement an OCR system for reading vehicle VIN numbers from digital photographs [1]. The ANN trained for this system proved to be very effective in identifying VIN numbers from processed sample images. By implementing a system to perform the image pre-processing done in [1], the trained OCR ANN would be capable of effective real-world performance. Dutta, Karmakar, and Si conducted a study that utilized different methods of training an ANN to diagnose patients based on their medical data with the end goal of creating an effective medical data mining tool [6]. The results of the study showed that the trained ANN managed an effective and balanced classification rate of the sampled medical data, demonstrating that

the approach used in [6] created a useful analysis tool for medical professionals.

2.2 Backpropagation

According to Negnevitsky, the conventional method for training feedforward MLP networks is the BP algorithm [2]. Many variants of BP have been devised since its inception, and include algorithms such as Levenberg-Marquardt, Bayesian regularization, and classical gradient descent [5]. BP works by calculating the error of the actual and desired outputs and propagating that error back through the network to adjust the ANN nodal weights. In general, many BP-based algorithms rely on an iterative machine learning technique called gradient descent [5]. The basic idea of gradient descent is that if the error function of a machine learning model is graphed, then all that needs to be done to optimize the model is to move along the graph to its lowest point. This approach is known as the steepest descent approach, and is the classical approach to most BP algorithms. The approach used in this study is the conjugate gradient approach. Nocedal and Wright give a comprehensive overview of the conjugate gradient approach in their book on numerical optimization approaches [11]. The conjugate gradient approach relies on the properties of conjugate vectors to find an optimal solution. It selects search vectors that are orthogonal to each other that move the search from the initial starting point to the optimal solution. Each move effectively moves from an initial point to the solution in one dimension. Thus, the conjugate gradient approach is guarenteed to converge in at most n steps, where n is equal to the number of dimensions of the optimization problem [11].

BP is relatively simple to implement in an artificial neural network given that it simply reverses the normal operation of a feedforward ANN. However, many BP algorithms have been consistently demonstrated to have several problems in terms of computational resources, time spent training, and a tendency to get stuck in local optima instead of finding global optima. BP is one of the oldest algorithms used for training artificial neural networks and is still used today due to how simple it is to implement. Many studies involving artificial neural

networks use BP as a comparison standard. Gudise and Venayagamoorthy conducted a study of BP and PSO in training an ANN to mimic a simple equation [5], and Dutta, Karmakar, and Si conducted a study of the Fireworks Algorithm, PSO, and BP with training an ANN to handle medical data [6]. Both of these studies found that BP algorithms underperformed when compared to search algorithms.

2.3 Particle Swarm Optimization

The PSO algorithm was first proposed by Kennedy and Eberhart in 1995 as a method of optimizing non-linear functions such as the weights of an artificial neural network [8]. The PSO algorithm is based off of the social behaviors of flocking birds and schools of fish, simulating their behavior with a swarm of particles. The PSO algorithm operates by randomly initializing a swarm of particles within a solution space bounded by the range of the optimization problem's dimensions. Each particle has its own position and velocity associated with it, and flies around the solution space looking for an optimal solution. The particles remember both their personal best position and the group's best position, and use those solutions to guide their movements. The two main equations that govern the particle movements are shown below:

$$v_p(i+1) = \omega * v_p(i) + c_1 * rand() * (p_{best_p} - x_p(i)) + c_2 * rand() * (g_{best} - x_p(i)) \tag{2.1}$$

$$x_p(i+1) = x_p(i) + v_p(i+1) \tag{2.2}$$

where v_p is the velocity of the particle being updated, x_p is the particle position, c_1 and c_2 are both positive constants, g_{best} is the global best particle position, p_{best} is the best position of the particle being updated, ω is the inertial decay constant, and $rand()$ is a number randomly drawn from a uniform distribution in the range of $[0, 1]$.

The c_1 and c_2 constants respectively represent the cognitive and social weights, and control how much the individual best solution and the group best solution influence how the

particles move through the search space. As the particle velocities are iteratively updated, they will fly towards the p_{best} and g_{best} values already recorded. The random coefficients in the velocity equation will allow more thorough exploration of the solution space by causing particles to "overshoot" these best values, investigating the solution space around the best positions. In the context of neural network training, each particle will represent an artificial neural network nodal weight configuration. Essentially, the PSO algorithm will train as many artificial neural networks as there are particles in the swarm.

PSO has been used to train artificial neural networks ever since it was proposed because it can efficiently optimize non-linear problems. In fact, one of the applications that Kennedy and Eberhart tested the PSO algorithm on in their original PSO paper was training simpler ANN [8]. A paper written by Gudise and Venayagamoorthy demonstrates how well PSO can train an ANN in comparison with BP algorithms [5]. The study found that PSO generally required less computations than BP did in training an ANN on a non-linear function, and that PSO converged on better solutions than BP. The study shows that PSO is well suited for artificial neural networks, but that an optimal setup for training artificial neural networks is required. Another study by Suresh, Harish, and Radhika compared PSO to BP in training an ANN to predict a patient's length of stay in a hospital [12]. In the ANN training trials conducted in [12], the PSO algorithm generally converged to better solutions than the BP algorithm that was used. Not only that, but the PSO algorithm generally converged faster than the BP algorithm. According to Suresh, Harish, and Radhika, the heuristic nature of PSO gives the algorithm an advantage over BP algorithms in optimization problems with a high number of local minima [12], a situation that tends to be common in training complex ANN.

2.4 Cooperative Particle Swarm Optimization

The CPSO algorithm was proposed by Van den Bergh and Engelbrecht in 2000 as an alternative to the PSO algorithm in training large artificial neural networks [4]. This

algorithm implemented cooperative learning strategies into the PSO algorithm, splitting the overall optimization task into a grouping of smaller particle swarms. The main advantage that the CPSO algorithm has over the original PSO algorithm is outlined in [7], which demonstrated that a neural network with a structure of 784 input nodes, 100 hidden nodes, and 10 output nodes forces the training algorithm to optimize about 79,510 different nodal weights. The PSO algorithm struggles to separate patterns in both source data and classification performance for effectively training an ANN when working in such high dimensions.

The CPSO algorithm works by optimizing a single vector of dimensions referred to as the context vector. This context vector is broken up into K parts, with each part getting a sub-swarm assigned to it. Each sub-swarm runs the PSO algorithm for one iteration. At the end of this iteration, the g_{best} solution of the sub-swarm particles are combined with the g_{best} solution(s) of the other sub-swarm(s) to reform the context vector for evaluation. This is repeated for each sub-swarm in the algorithm until all sub-swarms have been updated, completing one training iteration of the CPSO algorithm. By focusing particle swarms on sections of the context vector, each swarm has the advantage of working in less dimensions. If the context vector formed by the new sub-swarm g_{best} solutions is found to be a better solution than the current best context vector, the fitness value of every sub-swarm g_{best} particle is updated with the context vector's fitness value. The CPSO algorithm does this as a credit sharing approach to its cooperative learning nature. Since each sub-swarm g_{best} particle was a part of this better solution, they receive equal credit. This credit sharing mechanism guides the sub-swarms learning by allowing each sub-swarm to communicate with the other sub-swarm(s), allowing them to coordinate their search.

There are a variety of dimension splitting schemes for the CPSO algorithm [4, 7], but of interest to this study are the Lsplit and Esplit architectures [4]. The Lsplit architecture splits the context vector into sub-swarms based on the **layers** of the ANN. The weight matrix for the first layer to the second layer has a sub-swarm, the second layer to the third layer has a sub-swarm, and so on. The performance of the CPSO algorithm suffers when

correlated variables are split between the different sub-swarms that it uses, as each sub-swarm loses out on valuable information that would otherwise help them effectively optimize a problem [4]. When the CPSO algorithm is used to train an ANN, it becomes extremely difficult to ascertain if any specific nodal weights are interdependent on other nodal weights, let alone determining the actual interdependent nodal weights. The Lsplit architecture was created for training ANN to avoid this problem by splitting the context vector into sub-swarms based on the ANN layers. The nodal weights between layers are already grouped together in ANN theory, which implies a certain amount of interdependence between them. Splitting the context vector by the ANN layers not only minimizes the risk of splitting correlated variables between sub-swarms, but allows the nodal weights of layers to have separate ranges [4]. For example, the Lsplit architecture allows for the input to hidden layer nodal weights to have a different range than the hidden to output nodal weights.

The Esplit architecture **evenly splits** the context vector into two sub-swarms. The Esplit architecture was created to compensate for the Lsplit architecture splitting the context vector into a significantly skewed split of the context vector dimensions. An example of such a skewed split of dimensions is in a two swarm CPSO algorithm, where one sub-swarm is responsible for 90% of the context vector dimensions, and the other sub-swarm is responsible for the remaining 10%. This kind of imbalance is fairly common with ANN that have large input layer sizes, as most of the context vector is concentrated in the input layer to hidden layer connections. If the Lsplit architecture is used in this situation, it sabotages the sub-swarm that is optimizing the majority of the dimensions by restricting its freedom to maneuver [4]. The ANN used in this study is designed for image classification, which requires a very large input layer to account for all of the pixels in the images. This ANN design necessity results in the variable imbalance that the Esplit architecture was created to handle, so it was decided that this study would use the Esplit CPSO architecture in simulations.

The CPSO algorithm is an augmentation of the PSO algorithm, giving the CPSO algorithm many of the strengths of the PSO algorithm while attempting to minimize its

weaknesses regarding high solution space dimensionality. Bergh and Engelbrecht tested the PSO algorithm against a variety of CPSO architectures in training an ANN to classify the Iris, Glass, and Ionosphere machine learning data sets [4]. In each test, a CPSO architecture was able to achieve a better performance than the classical PSO algorithm did. The CPSO algorithm adds additional complexity to the PSO algorithm, but that complexity can easily boost the performance of the CPSO algorithm given an intelligently designed dimension splitting scheme and credit sharing strategy. Rakitianskaia and Engelbrecht examined the properties of the CPSO algorithm in training an ANN for classifying MNIST database [7]. They examined the effects of different activation functions and search space boundaries on fitness values and particle swarm diversity. The ANN trained by CPSO in [7] did not classify the MNIST database very well; however, the study clarified some of the major hurdles that the PSO algorithm faced in training a larger ANN. In particular, PSO has problems with saturating the ANN nodal weights during the training process. Nodal weight saturation is a situation that can occur during ANN training, where the nodal weights of a node can get large enough where the training algorithm cannot tell the difference between their output values. For example, the output of a sigmoid activation function is very similar for nodal weights of 40 and 400. Rakitianskaia and Engelbrecht showed that careful selection of activation functions and ANN structures is important to the ability of the PSO and CPSO algorithms to converge to a solution [7].

2.5 Bare Bones Fireworks Algorithm

The BBFWA was proposed by Tan and Li in 2017 as a simpler form of the Fireworks Algorithm [9]. The BBFWA mimics the explosion of a firework to explore a solution space. A firework is set in the solution space and then "explodes", scattering sparks within a predetermined explosion radius. The position of each spark is evaluated by the algorithm, and the best solution becomes the firework for the next iteration regardless if that solution is a spark or firework. The BBFWA relies on a single firework, and generates the same

amount of sparks at each iteration. The algorithm searches the solution space by varying the explosion radius of the firework, increasing or decreasing the firework explosion radius for better exploration or exploitation of the solution space, respectively.

The BBFWA starts by randomly initializing a firework within the solution space. In this study, the firework was initialized with values randomly drawn from a uniform distribution in the range of $[-0.12,\ 0.12]$ of the starting position. The limits of this range were set to balance the need to keep the ANN weights small, but still allow for variability in the simulation trials. The value of 0.12 was used as a common ANN initialization value that satisfied both of these concerns. The BBFWA then sets the explosion amplitude of this initial firework to the limits of the solution space. The equation for setting the initial firework explosion radius is shown below:

$$A_{CF} = \frac{Ub - Lb}{2} \tag{2.3}$$

where A_{CF} is the explosion amplitude of the core firework, Ub is the upper boundary of the solution space, and Lb is the lower boundary of the solution space. The original A_{CF} equation used in [9] is shown below:

$$A_{CF} = Ub - Lb \tag{2.4}$$

Note that the A_{CF} calculated in Equation (2.4) represents the diameter of the explosion amplitude along any dimension. The radius of the core firework's explosion amplitude is more useful for generating the initial firework explosion, as it sets the maximum location of a spark along any dimension to the bounds of the search space. Equation (2.3) represents the maximum possible explosion amplitude radius, based on the solution space boundaries. This is used to set the initial firework explosion amplitude to the optimum range for exploration of the solution space at the start of the BBFWA execution in this study.

Sparks are randomly generated from a uniform distribution in the range of $(x - A_{CF}, x + A_{CF})$. Any sparks that exceed the solution space boundaries are replaced with

new sparks randomly generated within the solution space. Each spark is evaluated using the search algorithm's fitness function. If one of the sparks has a better fitness than the firework, it takes the place of the firework for the next firework explosion. The explosion amplitude of the firework is adjusted based on the quality of the solutions found by its sparks. If the generated sparks find a better solution than the firework, the explosion amplitude will be multiplied by a reduction coefficient $C_r < 1$ to concentrate the sparks on the better solution. Otherwise, the firework location remains the same and the explosion amplitude is multiplied by an amplification coefficient $C_a > 1$ in order to find a better region in the search space.

While this study's testing and simulations make use of the BBFWA, the BBFWA was invented only a few years old ago, and thus there was a lack of concrete studies and related literature that apply the BBFWA to ANN research. It may be more beneficial to focus on the original Fireworks Algorithm, as the BBFWA preserves the essential functions and advantages of the original algorithm. Additionally, the BBFWA was demonstrated to be superior to the original algorithm and many of its variants in the original proposal paper for the BBFWA [9]. The original Fireworks Algorithm was proposed by Tan and Zhu in 2010 [9], making the Fireworks Algorithm a relatively new search algorithm. However, the applicability of the Fireworks Algorithm and its variants to ANN training problems has been thoroughly explored.

There is some experimental evidence to show that the Fireworks Algorithm could perform better than PSO or BP in training an ANN to classify real-world data. Dutta, Karmakar, and Si studied how well the Fireworks Algorithm, PSO, and BP did in training an ANN to diagnose patients using well-known medical databases [6]. The study showed that the Fireworks Algorithm was able to outperform both the PSO and BP algorithms in terms of overall artificial neural network performance. While the data sets used in [6] were less complex than the data sets used in this study, the results reported in [6] show the potential that PSO and the Fireworks Algorithm have for providing better performance and accuracy over BP. Another study conducted by Bolaji, Ahmad, and Shola compared the performance of the Fireworks Algorithm against several other metaheuristic algorithms

in training an ANN to classify several well-known UCI machine learning databases [13]. The Fireworks Algorithm either outperformed or matched the performance of the other metaheuristic algorithms used in [13] across some very challenging classification data sets, such as the Glass and Diabetes data sets. This study demonstrates the Firework Algorithm is just as effective as other metaheuristic algorithms in ANN training, and can work effectively in very challenging ANN training conditions.

CHAPTER 3: DESIGN PROCEDURE

3.1 Data Pre-Processing

The MNIST handwritten character database was selected as the ANN training data set for this study. The MNIST database was first published by LeCun et al. in 1998 [14], and it has been a benchmark for image classification that is still widely used today [7]. It is composed of 70,000 labelled images, with 60,000 of those images reserved as training data and the remaining 10,000 images reserved for testing. Each picture is a 28×28 greyscale image, with the pixel values represented as integers in the range of $[0,\ 255]$. Each digit image is matched with an integer label encoding the digits from 0 to 9. Due to the limitations of the programming language used in this study, the digit 0 has been changed to use a label of 10 instead of its original label of 0. All of the images have been normalized to values between 0 and 1 by dividing each pixel value by 255, which approximates the standard practice of normalizing the training data to a mean of 0 and a standard deviation of 1.

Two sets of data were used in this study based on the MNIST database: the original, unaltered MNIST database and an MNIST database that was compressed via Principal Component Analysis (PCA) dimension reduction. According to Clemmensen et al., PCA is a well-known dimension reduction technique commonly used in data mining and machine learning problems [15]. PCA maps high-dimensional data onto lower dimensional hyperplanes, finding ways of representing complex data with a smaller amount of features. It does this through the use of eigenvectors that describe the data set being compressed. The eigenvectors are sorted based on their associated eigenvalues from largest to smallest, with the largest eigenvalues corresponding to the highest data variances and the smallest eigenvalues corresponding to the smallest data variances [15]. The dimensionality of the reduced data set is determined by the number of eigenvectors, or principal components, that are selected.

Typically, the amount of principal components used is constrained by the overall data set variance that they represent. The data set variance that a number of eigenvectors ex-

plains is determined by a cumulative sum of the eigenvalues that correspond to the sorted eigenvectors. By normalizing the eigenvalues by the overall number of eigenvalues and cumulatively summing them up, it becomes possible to observe the overall variance of the data set that a combination of principal components explains. Figure 3.1 is a plot of the MNIST database features and their cumulative explained data variances that was generated to help select an effective number of dimensions for the PCA compressed MNIST database. This graph gives a visual demonstration of having to balance a smaller data set size with losing overall data. For example, the 100 eigenvectors shown in Figure 3.1 account for approximately 70% of the overall variance, while the first 200 eigenvectors approximately account for 87% of the overall variance. Most applications of the PCA dimension reduction technique seek to preserve most of the overall data variance of the source data when compressing it. Typically, an application using PCA dimension reduction attempts to preserve at least 90% of the source data variance. The underlying assumption of removing some of the source data variance is that any features removed by this variance threshold are either noise or unimportant to the overarching data set patterns.

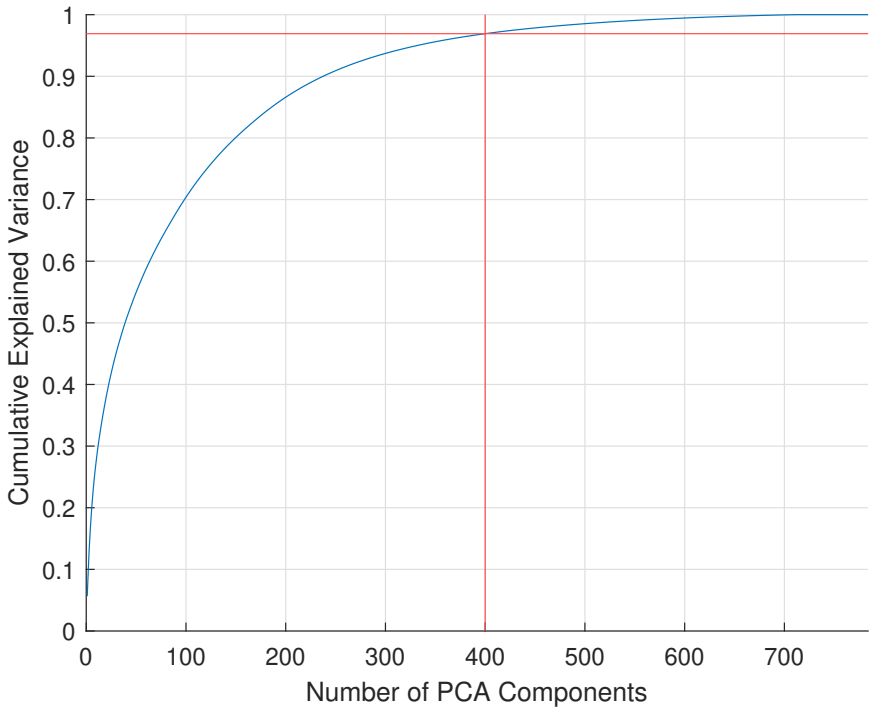

Figure 3.1: PCA Cumulative Data Variance by Principal Component

For this study, the number of dimensions that the MNIST database was compressed to represents approximately 97% of the overall variance of the original database. Additionally, it was considered important to this study that the number of dimensions that the MNIST database was compressed to still represented a square picture. For example, compressing the MNIST database to 144 features would have kept the images to a 12×12 shape, but compressing to 150 features would not have kept a recognizable image shape. The PCA compressed MNIST database was reduced from 784 features (28×28) to 400 features (20×20), which met both the variance condition with 96.92% overall variance explained and kept the images square. Figure 3.1 shows how the number of features and their overall variance were determined during the simulation design process. The vertical red line show where 400 principal components falls on the cumulative data variance curve for the MNIST database. The horizontal red line visually demonstrates the explained variance of 400 data features, showing that it is near the variance threshold of 97%.

The reason that PCA dimension reduction was used in this study is the high dimensionality of the MNIST database images. The topic of dimensionality has already been addressed in Section 2.4, but it is important to expand on it more in order to discuss some of the issues that this research encountered. Originally, this study had been planning to use the Canonical Fireworks Algorithm and Bare Bones Particle Swarm Optimization search algorithms for the ANN training simulations. However, the high dimensionality of the ANN structure, caused by the sheer number of data values in the MNIST 28×28 images, kept both algorithms from any form of reasonable performance. The use of PCA compressed images resulted in some improvement in the classification accuracy of the ANN trained by these algorithms. However, the number of dimensions that the MNIST database would have to be compressed to take advantage of this improvement in performace would have sacrificed most of the data variance in the images. Therefore, these algorithms were replaced with the CPSO algorithm, which was made to better handle high-dimensional optimization problems.

3.2 Artificial Neural Network Configuration

This study utilized a feedforward MLP ANN architecture for its simulations. The simulations used two different three layered MLP structures for the simulations, to account for the different dimensions of the two MNIST databases that were used in this study. The uncompressed MNIST ANN used a 784 input-35 hidden-10 output structure, and the PCA compressed MNIST ANN used a 400 input-35 hidden-10 output structure. The only difference in the two ANN structures used in the simulations was the size of the input layer; they both used the same activation functions and cost functions. The hidden layer used a Rectified Linear Unit (ReLU) activation function [16]. The general form of the ReLU equation is shown below:

$$f(x_i) = max(x, 0) = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ 0, & \text{if } x_i < 0 \end{cases} \tag{3.1}$$

The ReLU function is useful in gradient-based learning because it avoids vanishing gradients due to weight saturation. Weight saturation, in the context of ANN learning, occurs when a nodal weight becomes so large that any changes to it have little effect on the activation function. For example, the output of the sigmoid activation function [16] is very similar for an input value of 40 and 400. The learning of the PSO and CPSO algorithms were hampered by nodal weight saturation during development. The ReLU function was employed for the hidden layer of the neural networks in order to provide a fairer comparison between the three search algorithms tested in this study.

The activation function used in the ANN output layer was the softmax activation function [16]. The softmax activation function is used in multi-class classification problems to normalize the results of the output layer nodes to a range between 0 and 1, where all of the output numbers sum up to 1. The general formula for the softmax activation function is shown below:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.2}$$

This allows for the output layer's results to be interpreted as the probability that any one sample belongs to a class. A one-hot binary encoding scheme was used for the output layer, where the output node with the highest probability was assigned a 1 and all other nodes were assigned a 0.

The fitness function used by the search algorithms was the cross-entropy cost function [17]. The cross-entropy cost function allows for a bit-wise comparison between a ground truth label and a neural network label prediction, assuming a one-hot encoded binary scheme is used for both. The cost function assigns a logarithmically increasing penalty for bits that are 0 when they should be 1 as well as bits that are 1 when they should be 0. This function works well with classification ANN, as it can assign a cost to output values regardless of what label they actually represent. L2 regularization was also added to the fitness function. L2 regularization sums up the squared values of the nodal weights and adds the resulting value to the cost function. By doing this, L2 regularization encourages the ANN nodal weights to be small and reduces model complexity [18]. The cross-entropy formula with L2 regularization is shown below:

$$J = \frac{1}{m}\sum_{k=1}^{m}(-y_k \cdot \log(h_k) - (1-y_k) \cdot \log((1-h_k))) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ij}^{(l)})^2 \tag{3.3}$$

where J is the cost value of the current neural network predictions, m is the total number of data samples that have been analyzed, y_k is the ground truth label of a data sample, h_k is the actual neural network output for a data sample, λ is the regularization constant, L is the total number of layers in the ANN, s_l is the number of nodes in layer l, and $\theta_{ij}^{(l)}$ is the nodal weight matrix between layers l and $l+1$. The cross-entropy equation has been multiplied by -1 in order to ensure that the resulting value is positive, so that the search algorithms can properly minimize this equation. An important consideration to make regarding the

L2 regularization portion of Equation (3.3) is that the bias node connections should not be regularized. Given the constant output of the bias nodes, regularization of the bias nodes could cause problems for the ANN as it tries to adapt to the training data and desired behavior. Therefore, the summations of the nodes that are a part of the regularization term in Equation (3.3) are indexed from 1 instead of 0.

3.3 Simulation Setup

3.3.1 Search Algorithm Implementation

One of the most important elements in setting up the study simulations was understanding how to apply the search algorithms to the task of ANN training. The search algorithm search space consisted of nodal weights in the overall ANN configuration, meaning that each dimension represented a single nodal weight. In the context of this study, an ANN configuration refers to a complete set of nodal weights that creates an ANN. In the BBFWA and PSO algorithms, each particle or spark represented a single ANN configuration. This means that both algorithms effectively train a number of ANN equal to the size of their populations in this study. The CPSO algorithm splits an overall ANN configuration into parts and assigns each part a sub-swarm. While each portion of the ANN configuration has a number of different versions equal to the number of particles in its assigned sub-swarm, the CPSO only trains one full ANN configuration comprised of the best solutions of each sub-swarm.

The CPSO algorithm required a more detailed particle swarm initialization scheme than the BBFWA and PSO algorithms in setting up its initialization process. Determining the best solution for the BBFWA and PSO algorithms consisted of evaluating each particle with Equation 3.3. However, with the CPSO sub-swarm structure, determining the best overall ANN configuration requires combining each sub-swarm particle with particles from the other sub-swarm(s) in order to use Equation 3.3. In this study, particles were combined by index during this process. This means that particle 1 from sub-swarm A would be

evalauted with particle 1 from sub-swarm B, particle 2 from A with particle 2 from B, and so on until all particle pairs were evaluated. Since all CPSO sub-swarms had the same number of particles in them, this scheme worked fairly well at initializing the CPSO particles. This scheme was adopted to reduce the computational time and complexity of evaluating each sub-swarm particle with all other sub-swarm particles, a process that could become prohibitively expensive and time-consuming with larger particle populations and/or more sub-swarms.

3.3.2 Search Algorithm Parameters

The simulations in this study were run using MATLAB R2018b. This study augments research run by Gilley and Yan in a paper that compared search algorithms in improving upon a BP trained ANN [19]. Specifically, this study tests the BBFWA with a different set of parameters in an attempt to improve upon noted weaknesses with the original BBFWA parameters in [19]. [19] used a reduced set of sparks in the BBFWA that put the algorithm at a disadvantage to the other search algorithms. This study used an increased spark count and changed the c_r coefficient to attempt to further improve upon the performance of the BBFWA. For the purposes of this study, the BBFWA parameter set from [19] will be referred to as BBFWA Set 1, and the adjusted BBFWA parameters used in this study will be referred to as BBFWA Set 2. The parameters for each algorithm were set as follows:

1. **BBFWA Parameter Set 1:**
    - Number of Sparks: 30
    - Amplification Coefficient c_a: 1.2
    - Reduction Coefficient c_r: 0.5
2. **BBFWA Parameter Set 2:**
    - Number of Sparks: 50
    - Amplification Coefficient c_a: 1.2

- Reduction Coefficient c_r: 0.25

3. **PSO Parameters:**

   - Swarm Size: 30
   - c_1 Weight: 1.325
   - c_2 Weight: 1.325
   - ω Weight: 0.7298

4. **CPSO Parameters:**

   - Number of Sub-Swarms: 2
   - Swarm Size: 30
   - c_1 Weight: 1.325
   - c_2 Weight: 1.325
   - ω Weight: 0.7298

Some of the parameters for the BBFWA were sourced from established literature, and others were determined experimentally. The c_a value of 1.2 was sourced from the BBFWA proposal paper [9]. The c_r values in both BBFWA parameter sets were both experimentally set. In developing the BBFWA implementation for this study, the c_r coefficient was modulated in an attempt to improve upon the performance of the algorithm. The c_r value of 0.5 in the first BBFWA parameter set was observed to result in more accurate ANN classification rates than higher c_r values. The c_r coefficient was then set at 0.25 for the second parameter set because it encouraged better fitness values from the BBFWA, which resulted in more reliable ANN training results. The spark count of 30 in the first set of BBFWA parameters was selected based on reducing the time per training iteration while allowing the BBFWA to still converge to a solution. It was increased to 50 in the second set of BBFWA parameters because the data recorded in [19] showed that a BBFWA spark count of 30 was too low for reliable performance.

The PSO and CPSO shared the same parameters, as the CPSO algorithm uses the same method as the PSO algorithm in the particle training process. The ω inertial weight of 0.7298 was sourced from a PSO hyperparameter configuration proposed by Clerc [20]. Originally, the c_1 and c_2 weights were experimentally determined by the same process as the BBFWA c_r coefficient was. Setting the c_1 and c_2 weights to 1.325 encouraged better ANN performance on the image data sets. As the Esplit architecture was used for the CPSO algorithm and there were only three layers in the ANN structures in this study, only two sub-swarms were used in the CPSO algorithm. The number of CPSO sub-swarms used in this study mirrors the same design decision made in the CPSO proposal paper, which used 2 sub-swarms with the Esplit architecture on the tested three layer ANN [4].

CHAPTER 4: RESULTS

This study tested the BBFWA, PSO, and CPSO algorithms on the standard MNIST database and the PCA compressed MNIST database. This section will organize the data into two different sections in order to present and discuss data related to each database separately. A third section will be devoted to discussing overarching trends between the results of the two databases. For the sake of clarity, most of the figures generated from the simulation results will be placed in Appendix A. Only a few figures will be shown in this chapter to illustrate some of the conclusions made from the simulation results.

4.1 Standard MNIST Database Results

The original, uncompressed MNIST database was comprised of 28×28 greyscale images, resulting in 784 data features per image. This resulted in a large ANN structure in order to accommodate all of these data features, creating a difficult environment for the search algorithms. The performance of the search algorithms on the original, unmodified MNIST database was evaluated based on the ANN classification accuracy on both the training and testing data sets. Additionally, the search algorithm's fitness data was analyzed to provide fitness curves to illustrate how the search algorithms learned over the course of the simulation trials. The Monte Carlo simulation approach used in this study provided 50 different copies of these metrics per search algorithm. The mean and standard deviation of both ANN training and testing classification accuracies are listed in Table 4.1. The BP ANN training and testing classification accuracies are also listed in Table 4.1 for reference.

The fitness values achieved by the search algorithms were recorded and used to generate fitness curves that visually show how the search algorithms learned. A fitness curve for a search algorithm is a plot that shows the output of a search algorithm's fitness function on each iteration it executes. This provides a visual demonstration of a search algorithm's progression from beginning to end. The fitness values shown in the fitness curves are the

output of Equation 3.3 for an iteration of the search algorithm that the fitness curve belongs to. These fitness curves were augmented with error bars that demonstrate the 90^{th} and 10^{th} percentiles of the fitness values. The error bars show the general fitness curve trend of the search algorithms as they refined the BP ANN, without outlier data points from trials that had particularly good or bad random number draws. Each error bar plot also shows the average fitness curve as red dots on the plots. These average fitness curves were created by averaging all of the recorded fitness values at each iteration. Additionally, the BP cost curve was combined with the search algorithm error bar plots. The first 50 iterations on the plot show the cost curve from the BP ANN training process, which serves the same purpose as a fitness curve in this context. This shows where the BP training process started when training the original ANN, as well as where it ended. This point is where the search algorithms took over refining the ANN, attempting to further minimize the cross-entropy cost function from where the BP algorithm left off. The fitness curve error bar plots for BBFWA Set 1, BBFWA Set 2, PSO, and CPSO are shown as Figure 4.1 through Figure 4.4, as well as in Appendix A as Figure A.1 through Figure A.4.

The results shown in Table 4.1 demonstrate that the search algorithms were not able to make any significant improvements on the classification accuracy of the BP trained ANN. However, since the search algorithms were employed to fine-tune the pre-trained BP ANN, even smaller, more incremental improvements in the overall ANN classification accuracy have meaning in this study. On that basis, the BBFWA search algorithm performed the

Table 4.1: Mean and Standard Deviation of Standard MNIST ANN Classification Accuracies

Algorithm	**Training Accuracy (%)**	**Testing Accuracy (%)**
BP	89.8900	90.4100
BBFWA Set 1	89.9618 ± 0.3385	90.4366 ± 0.3536
BBFWA Set 2	89.8992 ± 0.2996	90.3916 ± 0.2756
PSO	89.7630 ± 0.1712	90.2138 ± 0.2398
CPSO	87.7485 ± 1.1946	88.3852 ± 1.1321

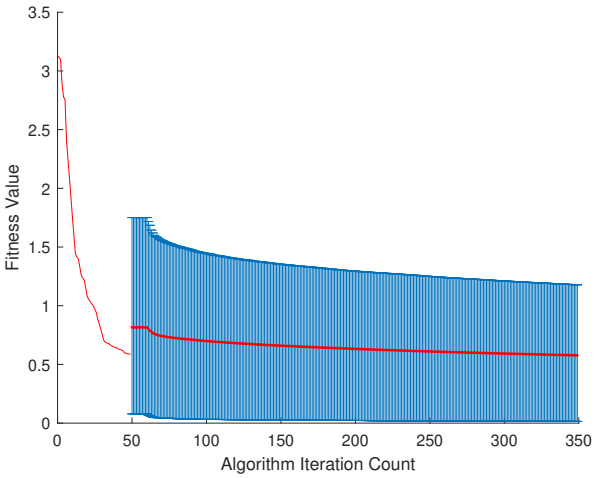

Figure 4.1: BBFWA Set 1 Standard MNIST Error Bar Fitness Curve Plot

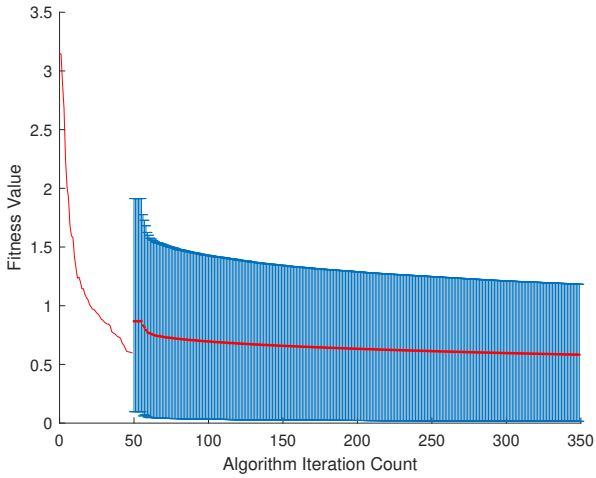

Figure 4.2: BBFWA Set 2 Standard MNIST Error Bar Fitness Curve Plot

best out of the three tested search algorithms. On average, both of the BBFWA parameter sets that were tested managed to either preserve the original ANN accuracy or make minor improvements to it. BBFWA Set 2 had a slightly smaller average accuracy than BBFWA Set 1, but it also had a smaller standard deviation. This would indicate that BBFWA Set 2 was more reliable than BBFWA Set 1, and that its average classification accuracy gives a better observation of its optimization performance. The difference between BBFWA Set 1 and BBFWA Set 2 is further demonstrated in Figure 4.1 and Figure 4.2. Both BBFWA sets stall for a certain amount of iterations as the firework's explosion radius calibrates to a range where the BBFWA can start improving upon its initial solution. However, BBFWA Set 2 generally takes fewer iterations than BBFWA Set 1 to start improving upon the pre-trained ANN. Additionally, the error bars in Figure 4.2 tend to be smaller than the error bars in Figure 4.1, reinforcing the difference in the standard deviations of the BBFWA parameter sets. Based on the results in Table 4.1 and the fitness curve error bar plots of BBFWA Set 1 and BBFWA Set 2, BBFWA Set 2 performed better than BBFWA Set 1 on the standard MNIST database.

The PSO algorithm did not perform as well as either of the BBFWA configurations. The PSO algorithm had the lowest standard deviation of the original MNIST trials, indicat-

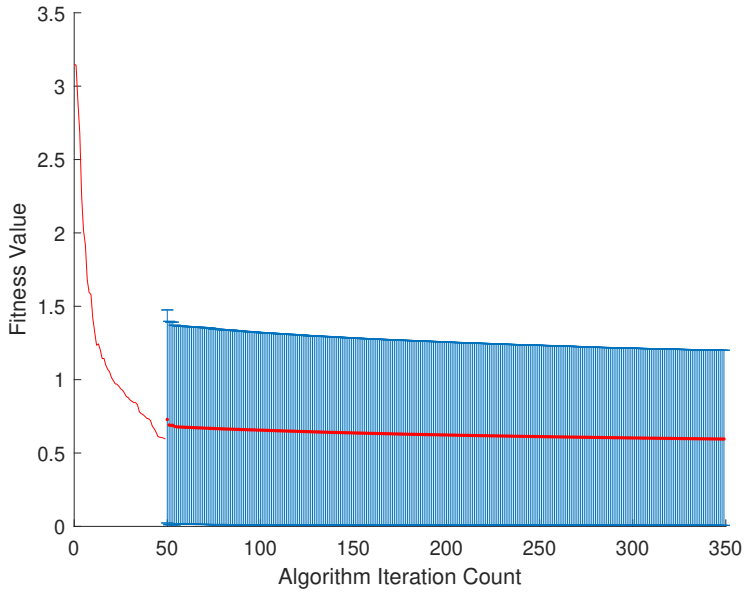

Figure 4.3: PSO Standard MNIST Error Bar Fitness Curve Plot

ing that it was the most reliable of the tested search algorithms. However, the average PSO classification accuracy was less than the pre-trained BP ANN accuracies were, indicating that the PSO algorithm on average caused a slight decrease in performance. In Figure 4.3, it can be observed that both the PSO algorithm's average fitness and error bar curves are very flat. This indicates that while the PSO algorithm was able to improve the pre-trained ANN, it was very slow in doing so. Compared to either of the BBFWA parameter sets, the PSO algorithm was slow and inefficient when working with the standard MNIST database.

The CPSO algorithm had the worst performance of the tested algorithms by far. Its average accuracies were significantly less than the pre-trained BP ANN's were, and the standard deviation of its results was significantly larger than the other search algorithms. This indicates that the CPSO algorithm badly under performed compared to the other algorithms. It had a tendency to decrease performance and it was very unreliable. The CPSO fitness curves shown in Figure 4.4 show that the CPSO barely improved on the pre-trained ANN. On average, the CPSO algorithm's fitness would only decrease for a few iterations, and then it would essentially lock at a single value for the rest of the simulation. The large standard deviation of the CPSO algorithm listed in Table 4.1 further shows how

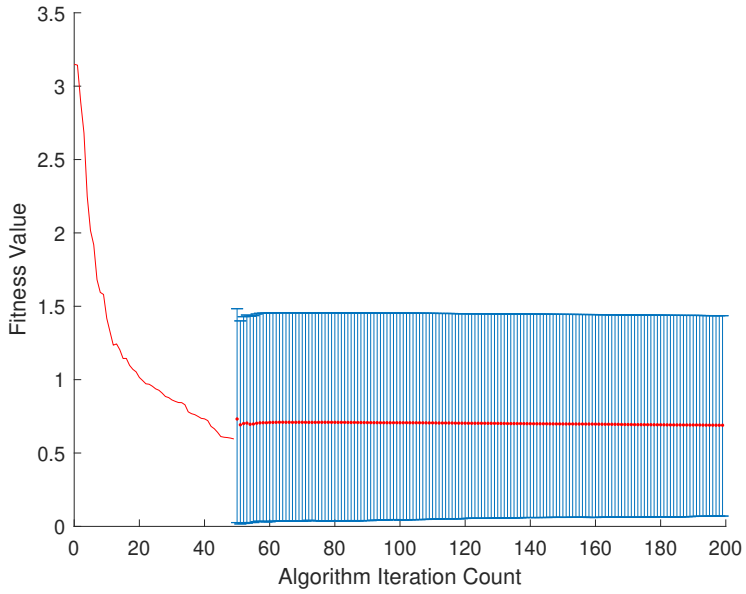

Figure 4.4: CPSO Standard MNIST Error Bar Fitness Curve Plot

poorly the CPSO algorithm performed on the standard MNIST database. The wide spread of ANN classification results suggests that the ANN being trained by the CPSO algorithm did not see any significant changes; rather, the ANN tended to stay at the classification accuracy it had when it was initialized. These results indicate that the CPSO algorithm was incapable of working with the standard MNIST database and the pre-trained ANN, given that its performance was more influenced by its random initialization than anything the algorithm did during the training process.

4.2 PCA MNIST Database Results

The PCA MNIST database is an altered version of the standard MNIST database, compressed from 784 data features per sample to 400 data features per sample via PCA dimension reduction. This had the effect of significantly reducing the dimensionality of the ANN optimization problem, making the simulation conditions more favorable for the PSO and CPSO algorithms. The PCA MNIST database was evaluated with the same methods and conditions as the standard MNIST database, and the same metrics were recorded. The mean and standard deviation of the refined ANN training and testing classification accuracies

are listed in Table 4.2. As with Table 4.1, the BP ANN training and testing classification accuracies are listed in Table 4.2 for reference. The fitness values of the search algorithms were recorded and used to generate the same type of fitness curve error bar plots as the ones created for Section 4.1. The fitness curve error bar plots for BBFWA Set 1, BBFWA Set 2, PSO, and CPSO are shown as Figure 4.5 through Figure 4.8, as well as in Appendix A as Figure A.5 through Figure A.8.

The data in Table 4.2 shows that three of the four search algorithms tested on the PCA MNIST database did not make any significant improvements on the performance of the pre-trained ANN. However, BBFWA Set 2 was able to improve upon the pre-trained ANN's training classification accuracy of the pre-trained ANN while preserving its testing classification accuracy. This shows that BBFWA Set 2 was superior to BBFWA Set 1 on the

Table 4.2: Mean and Standard Deviation of PCA MNIST ANN Classification Accuracies

Algorithm	**Training Accuracy (%)**	**Testing Accuracy (%)**
BP	90.0617	90.8500
BBFWA Set 1	89.9348 ± 0.3544	90.4724 ± 0.3489
BBFWA Set 2	90.3087 ± 0.2572	90.8064 ± 0.2722
PSO	90.0281 ± 0.1655	90.5500 ± 0.2353
CPSO	86.9439 ± 1.8804	87.5504 ± 1.8364

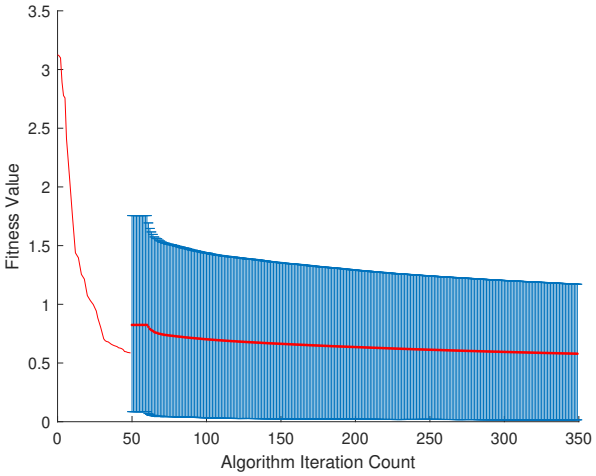

Figure 4.5: BBFWA Set 1 PCA MNIST Error Bar Fitness Curve Plot

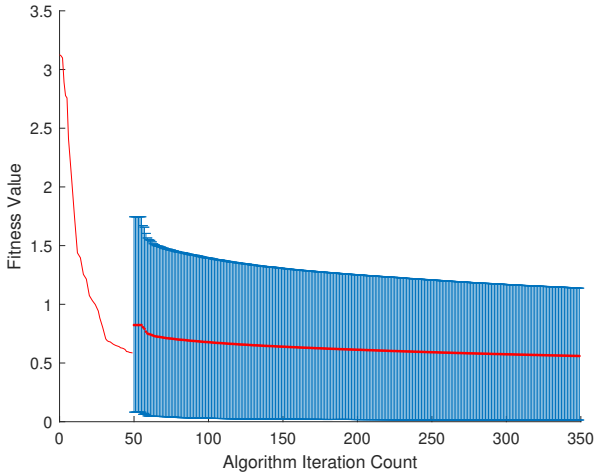

Figure 4.6: BBFWA Set 2 PCA MNIST Error Bar Fitness Curve Plot

PCA MNIST database. BBFWA Set 2 had a smaller standard deviation than BBFWA Set 1, while maintaining higher mean classification accuracies. Comparing the fitness curves of BBFWA Set 1 and BBFWA Set 2 in Figure 4.5 and Figure 4.6 shows the same characteristics that were noted in Section 4.1 when analyzing how the two BBFWA algorithm sets performed on the standard MNIST database. BBFWA Set 2 generally takes fewer iterations to start converging to a solution than BBFWA Set 1. Additionally, BBFWA Set 2 generally converges to a solution faster than BBFWA Set 1, given the steeper slope of its fitness curves and the smaller error bars in Figure 4.6.

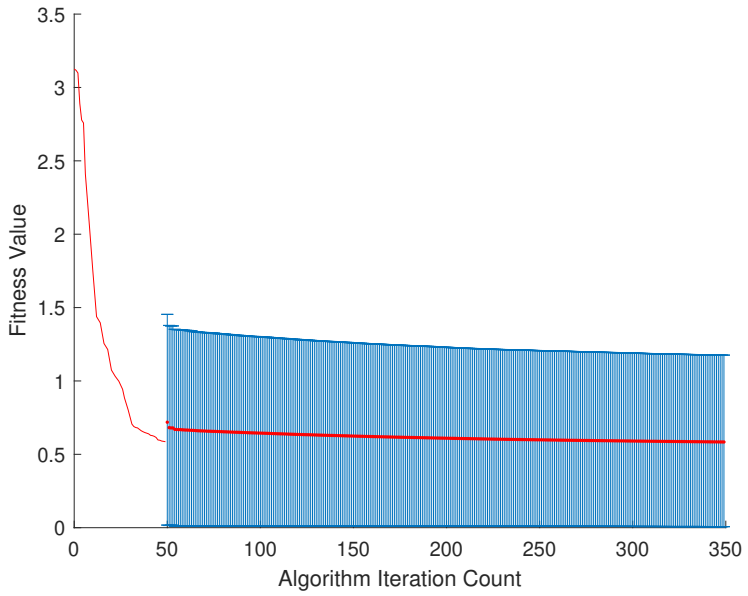

Figure 4.7: PSO PCA MNIST Error Bar Fitness Curve Plot

The PSO algorithm was the most reliable of all the tested search algorithms on the PCA MNIST database. Table 4.2 shows that the PSO algorithm had the lowest recorded standard deviation values of all the PCA MNIST database results. The PSO algorithm easily outperformed the CPSO algorithm and matched BBFWA Set 1 in terms of ANN classification accuracy mean and standard deviation. Deciding whether the PSO algorithm or BBFWA Set 1 performed better on the PCA MNIST database is more difficult than just comparing their performance metrics in Table 4.2. In [19], it was concluded that the BBFWA used in the paper performed better than the PSO and CPSO algorithms, despite

having slightly worse mean ANN classification accuracies and larger standard deviations than the PSO algorithm did. BBFWA Set 1 is the same BBFWA used in [19], and the data for everything in Table 4.2 with the exception of BBFWA Set 2 was sourced from the simulations run in [19]. The conclusion that BBFWA Set 1 was superior to the PSO algorithm in [19] was made on the basis that BBFWA Set 1 matched the performance of the PSO algorithm, even though the parameters used in BBFWA Set 1 put the algorithm at a significant handicap. This conclusion is supported by the results of BBFWA Set 2 in this study. BBFWA Set 2 used expanded parameters that improved its performance, and clearly performed better than any other search algorithm tested on the PCA MNIST database. Given that BBFWA Set 2 validated the conclusions of [19] where the BBFWA's performance was concerned, the PSO algorithm performed marginally better than BBFWA Set 1 on the PCA MNIST database in terms of ANN classification accuracy. In terms of the PSO algorithm's fitness curves, the same behavior noted in Section 4.1 occurs here. While the PSO algorithm constantly learns over the entire training process, it does so at a slower rate than either BBFWA algorithm. The PSO algorithm could probably reach the same effective solutions that BBFWA Set 2 did, but it would require a substantially increased number of training iterations to do so.

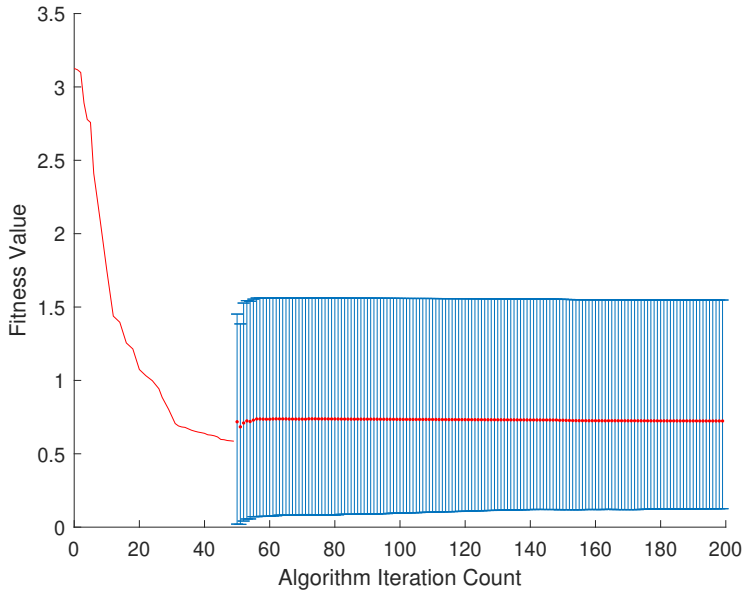

Figure 4.8: CPSO PCA MNIST Error Bar Fitness Curve Plot

The CPSO algorithm had the worst performance on the PCA MNIST database by far. There was a marked decline in both training and testing average ANN classification accuracies, and the standard deviations of both classification accuracies were much larger than the rest of the search algorithms. Figure 4.8 shows that the CPSO algorithm had many of the same problems on the PCA MNIST database that it had with the standard MNIST database. The CPSO algorithm had a tendency to lock into a constant fitness value after a few training iterations, showing its inability to learn with the PCA MNIST database. Comparing the fitness curves in Figure 4.4 and Figure 4.8 shows that the CPSO algorithm's problems were even more pronounced on the PCA MNIST database. The lower error bars in Figure 4.8 move towards the average fitness curve much faster than they do in Figure 4.4, showing that the CPSO algorithm was more unstable when working with the altered images and pre-trained ANN of the PCA MNIST database.

4.3 Overall Results Discussion

Considering the results presented in Section 4.1 and Section 4.2, the overall conclusion of this study is that the BBFWA Set 2 was the best performing search algorithm in the simulations. In terms of the ANN classification accuracies shown in Table 4.1 and Table 4.2, BBFWA Set 2 consistently produced classification accuracies that were higher than those of the pre-trained ANNs with a lower standard deviation. BBFWA Set 2 proved to be more effective than BBFWA Set 1 across both MNIST databases; BBFWA Set 2 took less iterations than BBFWA Set 1 to start the learning process, and its results were more consistent than BBFWA Set 1's were. Because BBFWA Set 2 took less iterations than BBFWA Set 1 did to start improving upon a pre-trained ANN, it was able to more effectively utilize the training iterations that it was allowed than BBFWA Set 1 was. As mentioned in Section 4.2, the conclusions of [19] hypothesized that expanded BBFWA parameters would lead to an increase in performance and set the BBFWA as the best search algorithm out of all the algorithms tested in the paper. The results of this study have verified this hypothesis, for the ANN

classification accuracies already mentioned and because of the fitness curves produced by the search algorithms. BBFWA Set 2 consistently has the steepest fitness curve in its starting iterations, and maintains the largest slope across the entire training process. This shows that it avoids the issues of BBFWA Set 1 in regards to the initial iteration stall as well as the issues of the PSO algorithm regarding how the flatness of its fitness curves show the algorithm's stagnation.

The PSO search algorithm stands as a close second to both of the BBFWA parameter sets. The consistently low classification accuracy standard deviations of the PSO algorithm shows that the PSO algorithm was the most reliable in its final results out of all the tested search algorithms. However, the PSO algorithm generally was unable to make any improvements upon either BP trained ANN. In fact, it had a tendency to cause a decrease in both BP trained ANNs' classification accuracies. The fitness curves in Figure 4.3 and Figure 4.7 give some insight into why the PSO algorithm behaved the way it did with the pre-trained ANN and MNIST databases. The difference between the PSO algorithm's initial and final fitness values tended to be very small, and the slope of its fitness curves tended to be just as small. This indicates that the PSO algorithm encountered significant difficulties in attempting to learn within the solution space with both versions of the MNIST database.

The best explanation for the PSO algorithm's results is that the algorithm struggled to work in the high dimensionality of the optimization problems presented by the standard and PCA MNIST databases. As already mentioned by Rakitianskaia and Engelbrecht, the PSO algorithm struggles to optimize problems with a large number of dimensions [7]. The slow rate of learning shown in the fitness curves indicates that the PSO algorithm was not able to find and pursue better solutions effectively. The recorded PSO algorithm's ANN classification accuracies show that the solutions the PSO particles chased were not necessarily improvements upon the original solution. This indicates that there were too many variables in the solution space for the particle swarm's communication mechanisms to guide the swarm to better solutions.

The CPSO algorithm was the worst performing search algorithm tested in this study. The high standard deviations and low means of the ANN classification accuracies produced by the CPSO algorithm demonstrate the how unreliable the CPSO algorithm was on both MNIST databases. In fact, the pre-trained ANN testing and training accuracies were over one standard deviation away from the mean accuracies of the CPSO algorithm with both MNIST databases, indicating that over 66% of the CPSO's refined ANN were significantly worse than the original ANN. The fitness curves produced from the CPSO algorithm's results on both the standard and PCA MNIST databases further illustrate the poor performance of the algorithm in this study. Both Figure 4.4 and Figure 4.8 show that the CPSO algorithm on average was unable to find any meaningful improvements on the pre-trained ANN. Even if there were a few trials that did manage to make some improvements on the pre-trained ANN, the vast majority of the other trials do not reflect this behavior.

The CPSO algorithm was the most complex search algorithm employed in this study in terms of its architecture. The number of sub-swarms used in the CPSO algorithm, as well as the context vector splitting scheme, play a significant role in how well the CPSO algorithm operates. Another consideration to make in CPSO architecture design is in how the search space variables are correlated with each other. If there are a set of variables that directly influence one another in an optimization problem, and those variables are split between CPSO sub-swarms, the overall performance of the CPSO algorithm will suffer [7]. One of the reasons for this is that if correlated variables are split between sub-swarms, those sub-swarms will miss out on important information as they optimize the variables that they are responsible for. In most ANN implementations, it is very difficult to determine which nodal weights correlate to other nodal weights, beyond the already existing ANN nodal layer structures. The poor performance of the CPSO algorithm in this study indicates that using the Esplit architecture with two sub-swarms may not have been an effective choice for this application.

The fitness curves of the CPSO algorithm in Figure 4.4 and Figure 4.8 show that the CPSO algorithm generally was unable to learn at all on either data set. The fitness data

used to generate Figure 4.4 and Figure 4.8 came from the fitness value history of the context vector. In the CPSO algorithm, each sub-swarm has its own set of fitness values that it tries to minimize. Whenever the sub-swarms improve upon the global solution, the sub-swarm particles that were responsible for the solution inherit the new fitness value. However, the sub-swarms do not directly minimize this value. This means that the overall fitness of the context vector has a tendency to increase and then stay constant at the beginning of the training process until the sub-swarms can lock into effective regions of the solution space and start improving the overall context vector. The CPSO algorithm was given more than enough training iterations to consistently avoid getting trapped in this kind of behavior. This behavior indicates that the CPSO sub-swarms were unable to find any regions where the two sub-swarms could work together to improve upon the context vector. Given the observed fitness curve behavior and that the BBFWA and PSO algorithms could find solution space regions to optimize the overall ANN configuration that they were given, the CPSO algorithm may be incompatible with the two-stage approach used in this study. Starting at a solution that a different algorithm arrived at deprives the CPSO algorithm of the freedom it needs to calibrate its sub-swarms to find better solutions.

An important consideration to make when analyzing the performance of any multi-class classifier is observing how the classifier handles the various data classes within its target data set. By analyzing the classification performance of the ANN trained and refined during the study simulations, misclassification trends can be identified in order to help determine why there are differences in classification rates between data classes. Additionally, this helps quantify the differences between a poor classifier and a great classifier. In order to perform this analysis on the thesis data, confusion matrices were generated for each search algorithm on both versions of the MNIST database by using the classification results of the training phase of all the ANN produced by the search algorithms. The training data results were used instead of the testing data results because the training phase of the ANN used 60,000, which provides more data to analyze than the ANN testing phase. These confusion matrices were generated for two test cases for each search algorithm tested in this study. The

test cases represented the best and worst training results of the search algorithms, which were determined to be the best and worst based on their respective final fitness values and associated training accuracies. The confusion matrices for the standard MNIST database results are shown in Appendix A as Figure A.9 through Figure A.16, and the PCA MNIST database confusion matrices are listed as Figure A.17 through Figure A.24. The BBFWA Set 2 confusion matrices for both the standard and PCA MNIST databases are included in this section as Figure 4.9 through Figure 4.12 as test cases representing the ANN classification behavior on their respective MNIST databases.

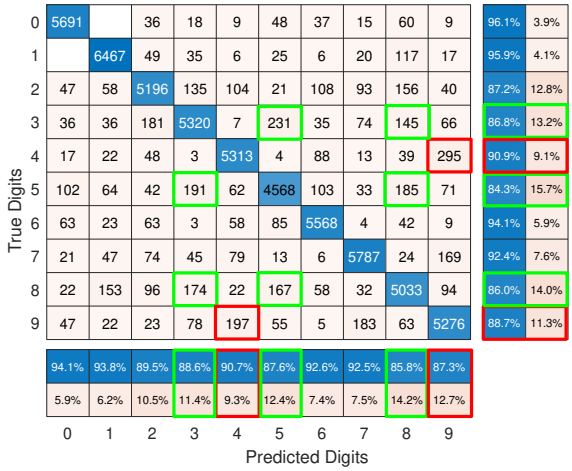

Figure 4.9: BBFWA Set 2 Standard MNIST Best Training Trial Confusion Matrix

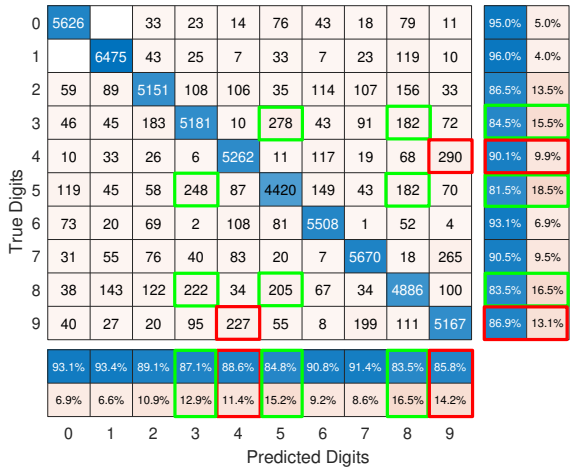

Figure 4.10: BBFWA Set 2 Standard MNIST Worst Training Trial Confusion Matrix

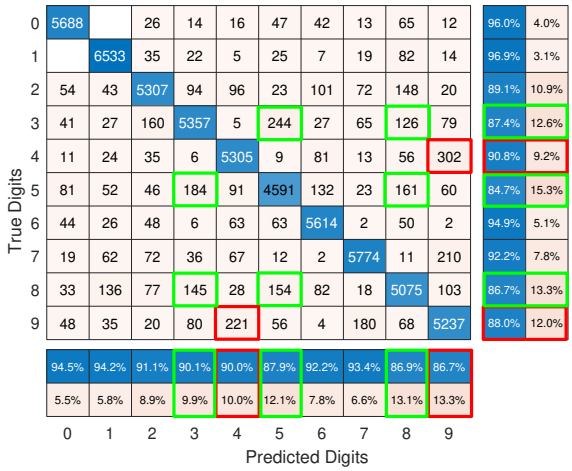

Figure 4.11: BBFWA Set 2 PCA MNIST Best Training Trial Confusion Matrix

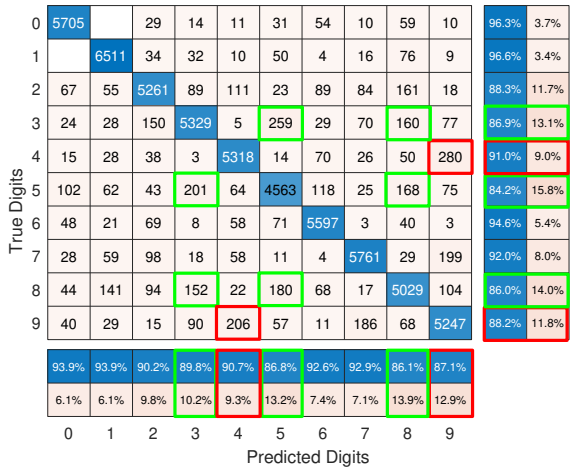

Figure 4.12: BBFWA Set 2 PCA MNIST Worst Training Trial Confusion Matrix

The search algorithms struggled with classifying some digits more than others. Figure 4.9 through Figure 4.12 use the same organizational scheme in showing data. Blue cells represent correct predictions, and orange cells show incorrect predictions. The percentages below the confusion matrices represent the percentage of digit predictions that were correct and incorrect. The percentages to the right of each confusion matrix represent the percentage of each digit class that was correctly and incorrectly classified. It can be observed from the confusion matrices that the refined ANN misclassified the digits 3, 4, 5, 8, and 9 the most. Digit 2 was somewhat of an issue for BBFWA Set 2, but comparing these confusion matrices to the rest of the confusion matrices in Appendix A shows that the digit 2 was not as common of an issue as the digit classes listed earlier. Determining which digit classes were commonly mistaken for other digit classes involved finding digit classes with high misclassification rates and then finding the highest prediction error number in the confusion matrix for that class. The ANN trained by the search algorithms had a tendency to confuse the digits 3, 5, and 8 with one another. The digits 4 and 9 were another group of digit classes that were commonly confused by the ANN. To help illustrate how the confusion matrices were used to arrive to this conclusion, all of the confusion matrices have highlighting around the relevant cells. Green highlighting was used to identify classification rates and prediction error cells that demonstrate that the ANN had difficulty distinguishing the digits 3, 5, and 8 from each other. Red highlighting was used to do the same task for digits 4 and 9.

As Figure 4.9 through Figure 4.12 show, the digits 3, 5, and 8 all had higher misclassification rates. If the orange cells for the class predictions or the digit class populations are compared for any one of these three digit classes, the errors for the other two classes in the group is always unusually higher than the rest of the prediction errors. The same goes for digits 4 and 9; the ANN classification errors for these classes have an even more noticeable tendency to favor the other class in this group than they did for the other digit class group. When these digits are written, they tend to look very similar to each other. The digits 4 and 9 differ by only the presence of straight lines and angles or curves on the top of each digit, which can blur together for many people's handwriting. The digits 3, 5, and 8 all share the

same type of curve on the lower half of the digit, and the digit 3 is written almost the same as digit 8. Since the trained ANN have a tendency to confuse all three digits with each other, it seems reasonable to conclude that the lower curve of all three digits is a key discriminative feature being learned by the ANN as they are refined by the search algorithms.

These two digit class groups are the biggest hurdle for the ANN being refined by the search algorithms. Comparing the best ANN training trial results in Figure 4.9 and Figure 4.11 to the worst training trial results in Figure 4.10 and Figure 4.12 clarifies the difference between good and bad ANN classifiers. The rates of correct and incorrect classification for digit classes 0, 1, 2, 6, and 7 tend to remain fairly constant between the best and worst training runs on either database. What really changes between the best and worst training runs is that the best training results significantly reduce the rate of misclassification of the other five digit classes. This clearly demonstrates that one of the main influencing factors in how well the handwritten character recognition ANN performs in classifying these handwritten digits is dependent on how well it can handle these difficult distinctions. The ANN used to create Figure 4.9 and Figure 4.11 has a higher success rate on all of the challenging digit classes than the ANN used to create Figure 4.10 and Figure 4.12. This is what resulted in the striking difference in performance between the selected trial results.

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1 Conclusions

This study compared the ability of the BBFWA, PSO, and CPSO search algorithms in refining an ANN trained with conjugate gradient BP to classify the MNIST database. After analyzing the results presented in Chapter 4, this study concludes that the BBFWA was the best performing search algorithm in producing a better ANN than it started with. The optimization problem of refining the BP trained ANN had a very high dimensionality, which was caused by the tens of thousands of nodal weights in the ANN structures used in this study. Additionally, conjugate gradient BP is a fairly efficient training algorithm in terms of reaching a good solution, and the ANN trained with it were already able to achieve a high classification performance on both the standard and PCA MNIST databases. Despite both of these major challenges, the BBFWA was able to reliably produce a better ANN than it started with.

The methods used for the BBFWA in this study explored some of the conclusions made by Gilley and Yan in [19]. The data and conclusions of [19] noted that the BBFWA setup it used had issues in both converging to a solution and the consistency of the algorithm's final solutions. BBFWA Set 1 was a full copy of the BBFWA used in [19], in order to allow this study to compare against the results in that paper. BBFWA Set 2 was an experimental parameter set for the search algorithm, created as an improvement to BBFWA Set 1. By increasing the number of sparks per explosion and by adjusting the reduction coefficient to cause the firework's explosion radius to contract faster, BBFWA Set 2 removed the handicap that BBFWA Set 1's low spark count caused and allowed the firework to zoom into the number range of the ANN nodal weights faster than BBFWA Set 1 could. BBFWA Set 2 clearly out performed BBFWA Set 1 on the PCA MNIST database, and produced more reliable results on the standard MNIST database than BBFWA Set 1 did. The data and fitness curves clearly show that the adjusted parameters in BBFWA Set 2 improved upon

the weaknesses of BBFWA Set 1. The number of iterations it took for the firework used by BBFWA Set 2 to calibrate its explosion radius to a range where it could start improving the BP ANN was smaller than the number of iterations it took BBFWA Set 1 to accomplish the same thing in every trial run in this study. On average, when BBFWA Set 2 started learning, it did so faster than BBFWA Set 1 on both MNIST databases. This allowed BBFWA Set 2 to use its time more efficiently than BBFWA Set 1. Additionally, BBFWA Set 2 always had a smaller classification accuracy standard deviation than BBFWA Set 1, showing that BBFWA Set 2 was always more consistent in its final results than BBFWA Set 1 was. This study concludes that the parameter changes made to BBFWA Set 1 to create BBFWA Set 2 were effective in improving the performance of the BBFWA for this application, which verifies the conclusions of [19] regarding the BBFWA search algorithm.

The PSO algorithm struggled to overcome the dimensionality issues presented by the problem of refining the BP trained ANN. The classification accuracies for both the standard and PCA MNIST databases were worse than the original ANN's were. It would seem that compressing the MNIST database using PCA dimension reduction techniques made little difference in how the PSO algorithm behaved in this study. The PSO algorithm's classification accuracy standard deviations for the PCA MNIST database were nearly identical to its classification accuracy standard deviations for the standard MNIST database. Additionally, the difference between the average PSO and BP classification accuracies is nearly identical. The fitness curves generated for the PSO algorithm for both data sets shows a very slow rate of convergence to a solution. The PSO algorithm was the most reliable algorithm when it came to being consistent in the solution it arrived at. However, the PSO algorithm was consistently very slow to learn and arrive at a solution, and the amount of dimensions it was working in seemed to significantly reduce the sensitivity of the PSO particle swarm to better solutions.

The CPSO algorithm was unable to make any improvements to the BP trained ANN in this study for either MNIST database. Part of the reason for the CPSO algorithm's inability to optimize the pre-trained ANN in this study may come from the dimension splitting

architecture used in this study. Only the Lsplit and Esplit architectures were investigated in this study using two sub-swarms, so a different architecture or an increased number of sub-swarms with the Esplit architecture may have resulted in some improvement in the CPSO algorithm's performance. However, any problems that may have arisen as a result of the CPSO dimension splitting architecture were over-shadowed by the problems that occurred as a result of the nature of the optimization problem itself. The CPSO algorithm will spend a number of training iterations finding solution space regions where its sub-swarms can start improving upon the context vector representing the overall optimization problem solution. Since the CPSO algorithm had to start at a solution that a different optimization algorithm arrived at, the CPSO sub-swarms did not have the ability to find their own regions to explore in order to improve the context vector solution. While there could have been individual trials where the CPSO algorithm may have made improvements to the pre-trained ANN, the CPSO fitness curves in Figure 4.4 and Figure 4.8 indicate that the majority of the CPSO trials were unable to. In fact, those fitness curves seem to indicate that the sub-swarms spent the entire training process trying to find effective regions for both sub-swarms, which reinforces the conclusion that the CPSO algorithm was not very suited to this application.

This study used two different input data sets based on the MNIST database: the standard, unaltered MNIST database and the PCA compressed MNIST database. The purpose of applying PCA dimension reduction techniques to the MNIST database was to reduce the dimensionality of the image data set and consequently the dimensionality of the ANN used to classify it. However, the results of the two different MNIST databases are not as different as they were expected to be. While the classification accuracies of the PCA MNIST results are higher than the standard MNIST classification results, the difference is mainly due to the fact that the BP trained ANN for the PCA MNIST had higher classification accuracies than the BP trained ANN for the standard MNIST. The behavior of the PSO and CPSO algorithms was fairly constant, turning out almost identical fitness curves and classification accuracy patterns. The BBFWA was the only algorithm that saw any noticeable change between the two data sets. BBFWA Set 1 and BBFWA Set 2 were closely matched on the

standard MNIST database in terms of classification accuracies. However, BBFWA Set 2 saw a noticeable increase in performance on the PCA MNIST database by increasing both the training and testing accuracy of the BP trained ANN. Conversely, BBFWA Set 1 saw a decrease in performance on the PCA MNIST database, with its classification accuracies decreasing to match the PSO algorithm's classification accuracies. Unfortunately, based on the results of this study, it would seem that applying PCA dimension reduction techniques to the MNIST database was not very effective in increasing the performance of the search algorithms.

5.2 Future Work

There are several avenues of investigation for future work on this research. The ANN structure and training process used in this study were limited by computational resources and simulation time constraints, and would benefit from the use of expanded parameters. The hidden layer of the ANN for both MNIST databases was rather small for the size of the input layer in either database. Additionally, the number of training iterations allocated to the search algorithms was also smaller than normal. Rerunning the study simulations with a larger hidden layer and much higher number of training iterations could yield different results and behaviors than the ones seen in this study. Another line of research would be a more thorough exploration of dimension reduction techniques. The PCA dimension reduction technique is robust, but it is an older technique that may be unintentionally sacrificing data important to the ANN classifiers even as it preserves the data variances of the original data. The difference between the PCA and standard MNIST databases was not as pronounced as it was hoped to be in this study. Therefore, testing other dimension reduction techniques along with PCA techniques in compressing the MNIST database to different dimensions could give some useful insight as to how those techniques could help search algorithms with such high dimension problems. Another interesting line of research would be investigating the CPSO algorithm's performance in more detail. The CPSO algorithm did not work

well with either database in this study, so it would be interesting to investigate how the CPSO algorithm handles optimizing pre-trained ANN in general to see if the problems that the CPSO algorithm had in this study were unique or not. Additionally, it would be worthwhile to try and determine some general guidelines for selecting a CPSO dimension splitting architecture of a problem, as well as the number of sub-swarms that should be used with it. The last line of research would be comparing the performance of the conjugate-gradient BP algorithm against the search algorithms in this paper. This would provide a more thorough comparison between the performance of the BP algorithm and the search algorithms. The results of the BBFWA indicate that it could viably compete with the BP algorithm in training an ANN on the MNIST database, as its performance in this study shows that it was not unduly impacted by the optimization problem's dimensionality unlike the PSO and CPSO algorithms.

BIBLIOGRAPHY

[1] Parul Shah, Sunil Karamchandani, Taskeen Nadkar, Nikita Gulechha, Kaushik Koli, and Ketan Lad, "Ocr-based chassis-number recognition using artificial neural networks," *2009 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, Nov 2009.

[2] Michael Negnevitsky, *Artificial Intelligence: A Guide to Intelligent Systems*, chapter Artificial Neural Networks, pp. 165–217, Addison-Wesley, Harlow, 2005.

[3] Donald Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Redwood City,CA, 2 edition, 1998.

[4] Frans van den Bergh and Andries Engelbrecht, "Cooperative learning in neural networks using particle swarm optimizers," *South African Computer Journal*, vol. 26, pp. 84–90, 01 2000.

[5] V. G. Gudise and G. K. Venayagamoorthy, "Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706)*, April 2003, pp. 110–117.

[6] Ram Kinkar Dutta, Nabin Kanti Karmakar, and Tapas Si, "Artificial neural network training using fireworks algorithm in medical data mining," *International Journal of Computer Applications*, vol. 137, no. 1, pp. 975–8887, 04 2016.

[7] A. Rakitianskaia and A. Engelbrecht, "Training high-dimensional neural networks with cooperative particle swarm optimiser," in *2014 International Joint Conference on Neural Networks (IJCNN)*, July 2014, pp. 4011–4018.

[8] James Kennedy and Russell Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, Nov 1995, vol. 4, pp. 1942–1948.

[9] Ying Tan and Junzhi Li, "The bare bones fireworks algorithm: A minimalist global optimizer," *Applied Soft Computing*, vol. 62, pp. 454–462, 2018.

[10] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[11] Jorge Nocedal and J Stephen Wright, *Numerical Optimization*, Springer-Verlag GmbH, second edition, 2006.

[12] Abhijit Suresh, K.V. Harish, and N. Radhika, "Particle swarm optimization over back propagation neural network for length of stay prediction," *Procedia Computer Science*, vol. 46, pp. 268 – 275, 2015, Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[13] Asaju Bolaji, Ali Aminu Ahmad, and Peter Shola, "Training of neural network for pattern classification using fireworks algorithm:," *International Journal of System Assurance Engineering and Management*, vol. 9, 07 2016.

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[15] Sara Sharifzadeh, Ali Ghodsi, Line H. Clemmensen, and Bjarne K. Ersbøll, "Sparse supervised principal component analysis (sspca) for dimension reduction and variable selection," *Engineering Applications of Artificial Intelligence*, vol. 65, pp. 168 – 177, 2017.

[16] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *CoRR*, vol. abs/1811.03378, 2018.

[17] G.E. Nasr, E. Badr, and C. Joun, "Cross entropy error function in neural networks: Forecasting gasoline demand.," in *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, Jan. 2002, pp. 381–384.

[18] Andrew Ng, "Feature selection, l 1 vs. l 2 regularization, and rotational invariance," *Proceedings of the Twenty-First International Conference on Machine Learning*, 09 2004.

[19] Patrik Gilley and Yanjun Yan, "Comparison of search optimization algorithms in two-stage artificial neural network training for handwritten digits recognition," in *2020 SoutheastCon*. IEEE, 2020.

[20] M. Clerc, "The swarm and the queen: towards a deterministic and adaptive particle swarm optimization," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, July 1999, vol. 3, pp. 1951–1957 Vol. 3.

APPENDIX A: COLLECTED DATA

A.1 Search Algorithm Error Bar Fitness Curves

A.1.1 Standard MNIST Database Fitness Curves

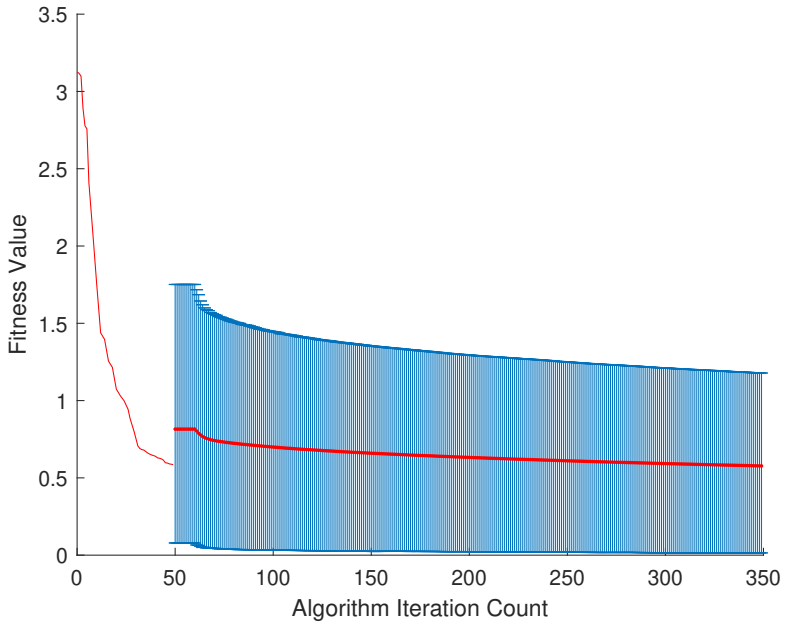

Figure A.1: BBFWA Set 1 Standard MNIST Error Bar Fitness Curve Plot

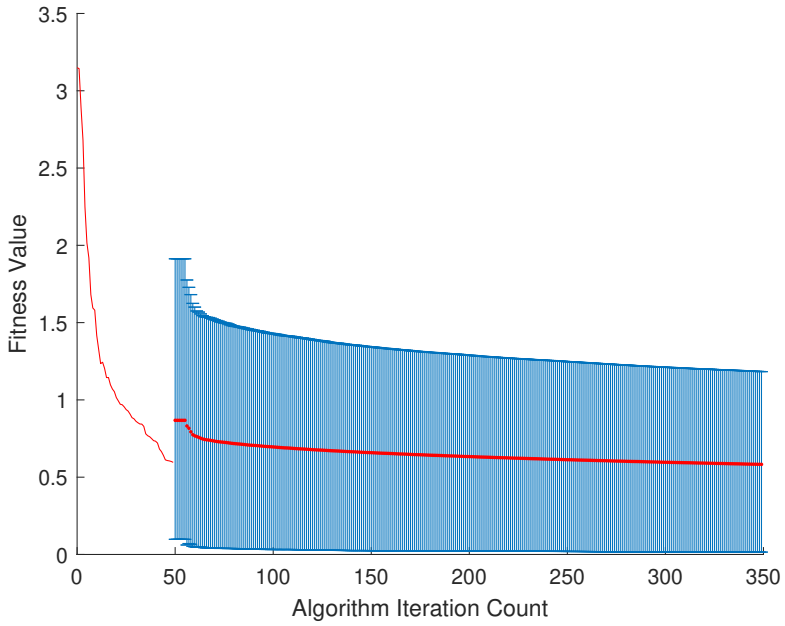

Figure A.2: BBFWA Set 2 Standard MNIST Error Bar Fitness Curve Plot

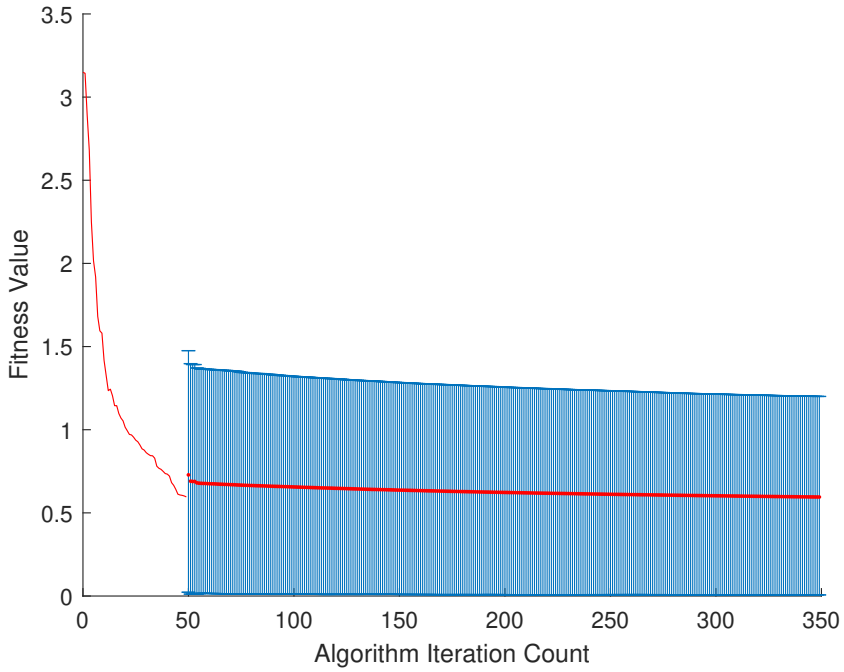

Figure A.3: PSO Standard MNIST Error Bar Fitness Curve Plot

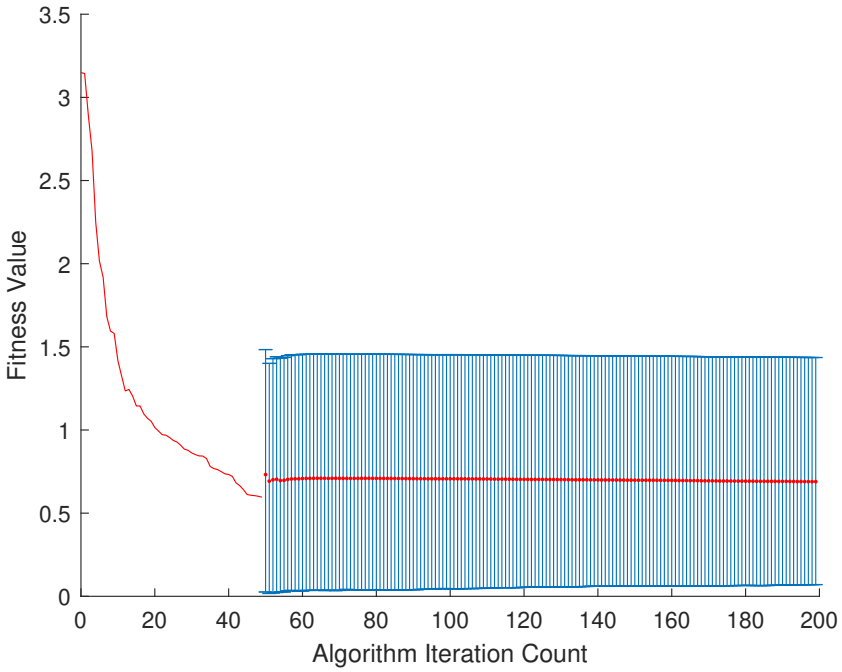

Figure A.4: CPSO Standard MNIST Error Bar Fitness Curve Plot

A.1.2 PCA MNIST Database Fitness Curves

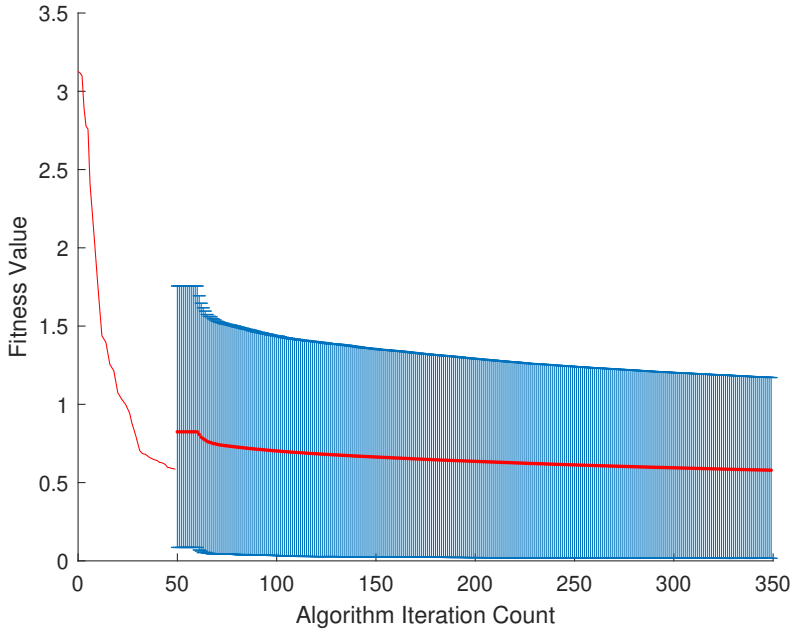

Figure A.5: BBFWA Set 1 PCA MNIST Error Bar Fitness Curve Plot

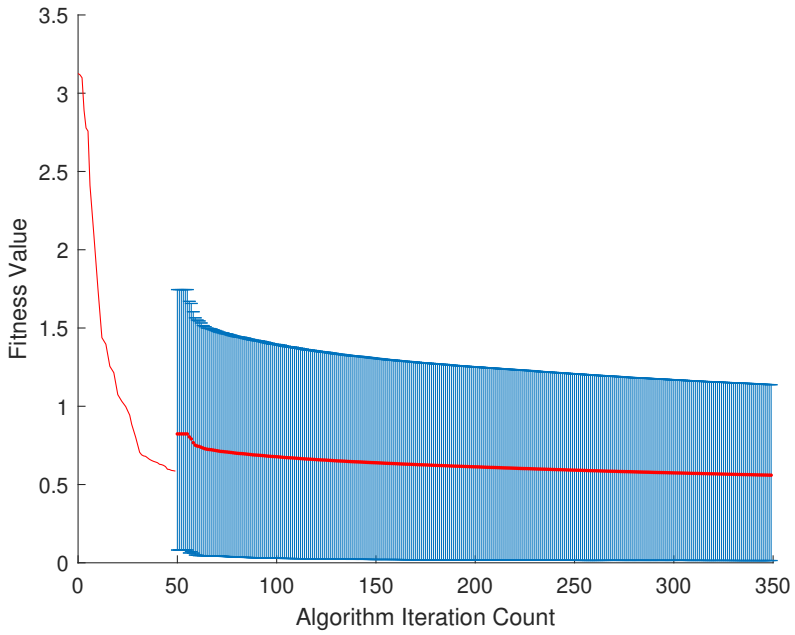

Figure A.6: BBFWA Set 2 PCA MNIST Error Bar Fitness Curve Plot

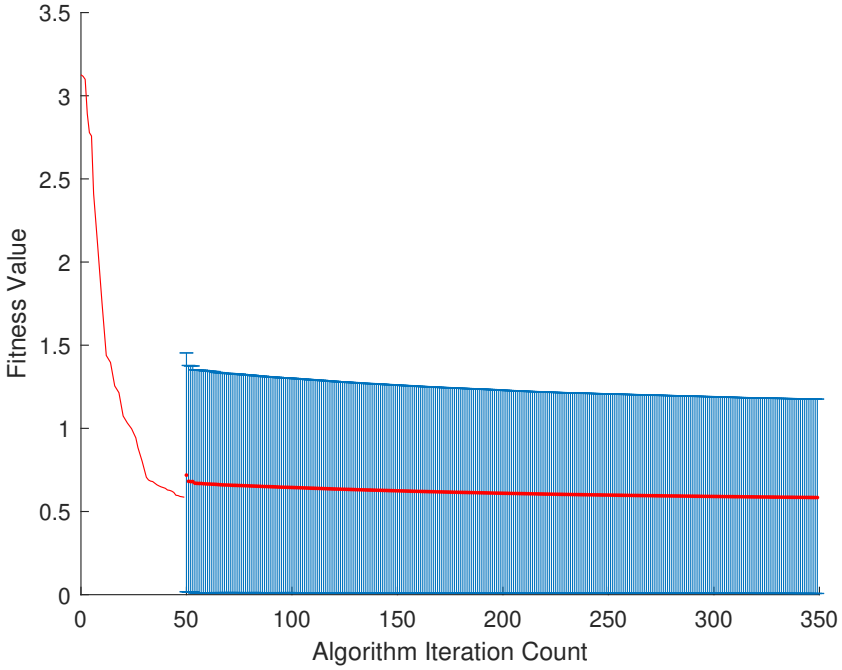

Figure A.7: PSO PCA MNIST Error Bar Fitness Curve Plot

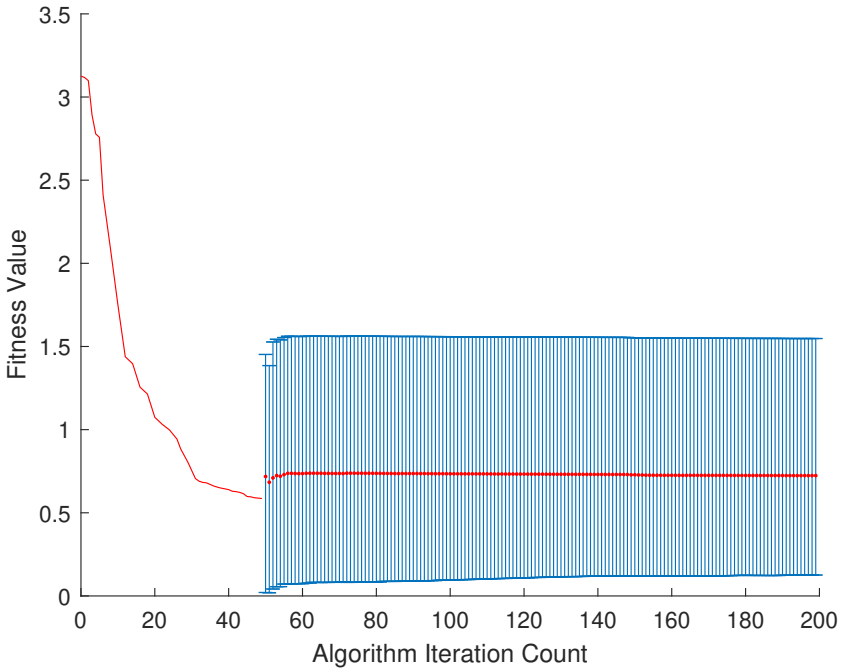

Figure A.8: CPSO PCA MNIST Error Bar Fitness Curve Plot

A.2 Search Algorithm Confusion Matrices

A.2.1 Standard MNIST Database Confusion Matrices

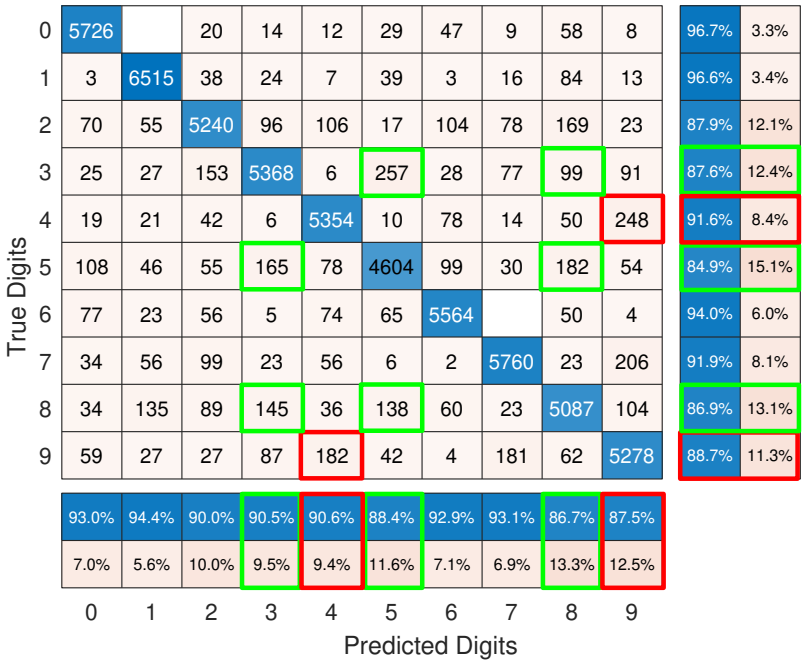

Figure A.9: BBFWA Set 1 Standard MNIST Best Training Trial Confusion Matrix

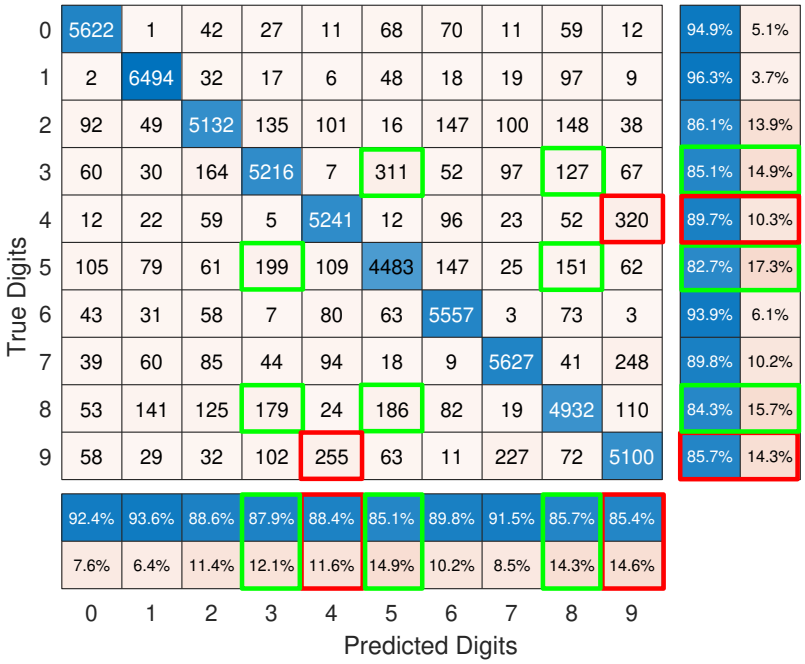

Figure A.10: BBFWA Set 1 Standard MNIST Worst Training Trial Confusion Matrix

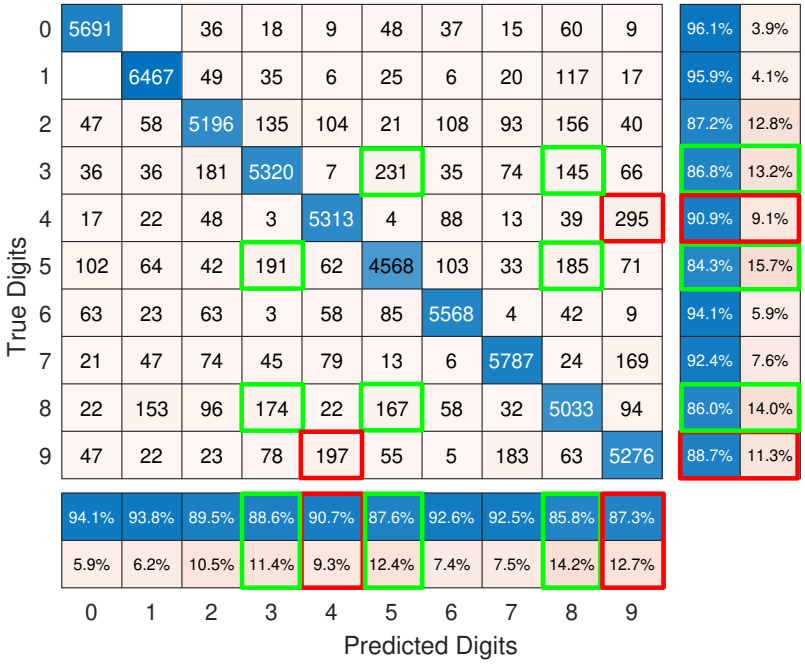

Figure A.11: BBFWA Set 2 Standard MNIST Best Training Trial Confusion Matrix

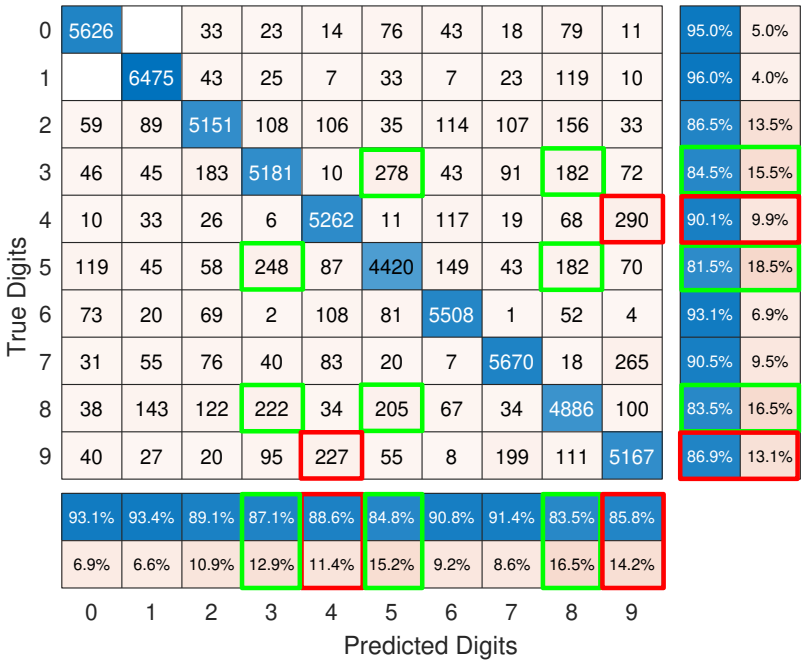

Figure A.12: BBFWA Set 2 Standard MNIST Worst Training Trial Confusion Matrix

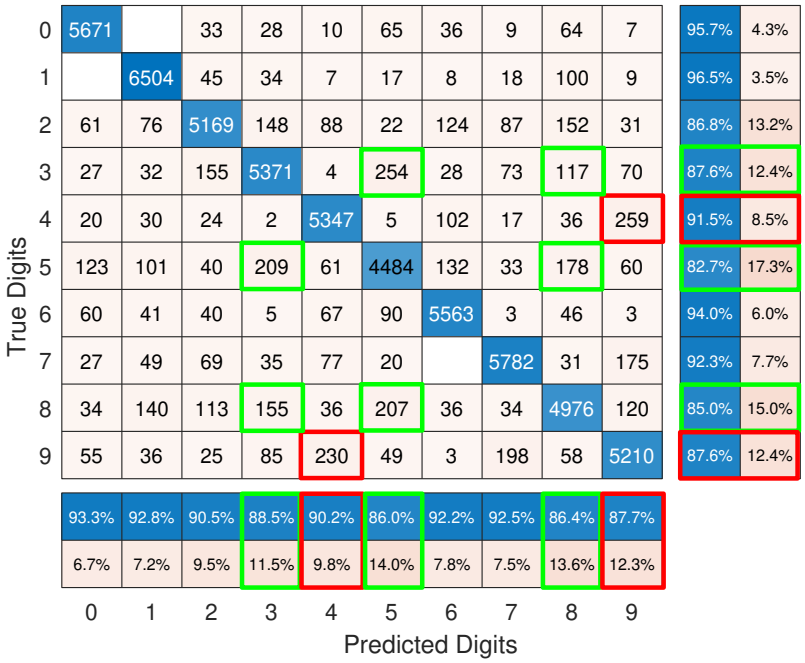

Figure A.13: PSO Standard MNIST Best Training Trial Confusion Matrix

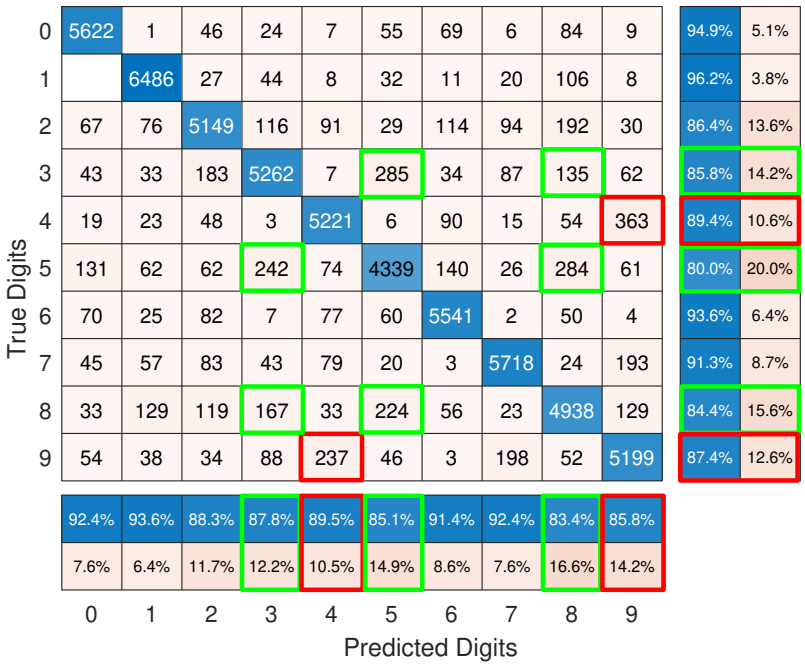

Figure A.14: PSO Standard MNIST Worst Training Trial Confusion Matrix

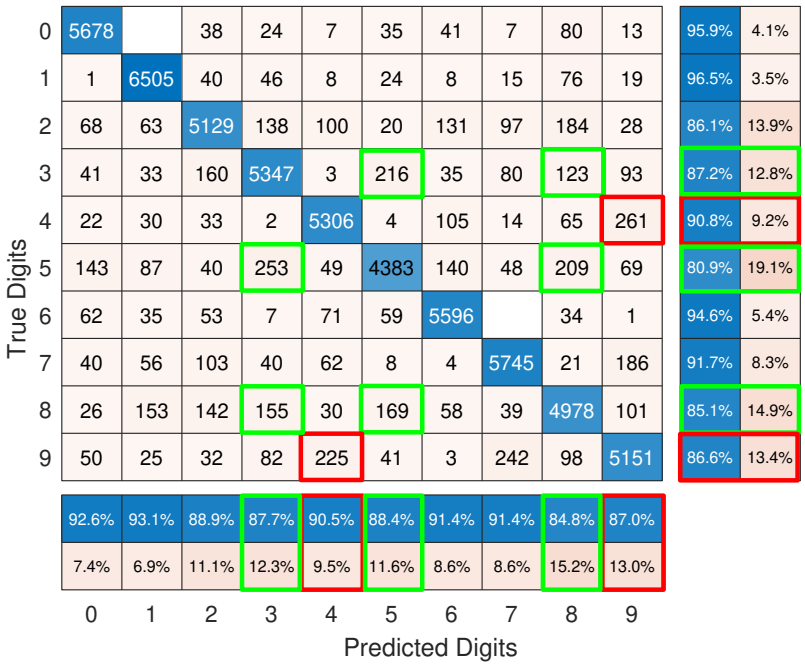

Figure A.15: CPSO Standard MNIST Best Training Trial Confusion Matrix

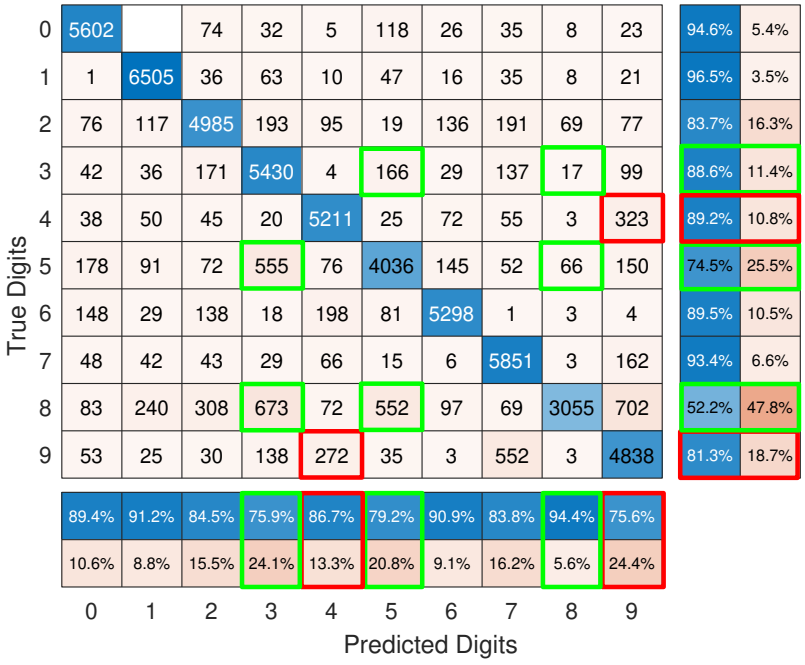

Figure A.16: CPSO Standard MNIST Worst Training Trial Confusion Matrix

A.2.2 PCA MNIST Database Confusion Matrices

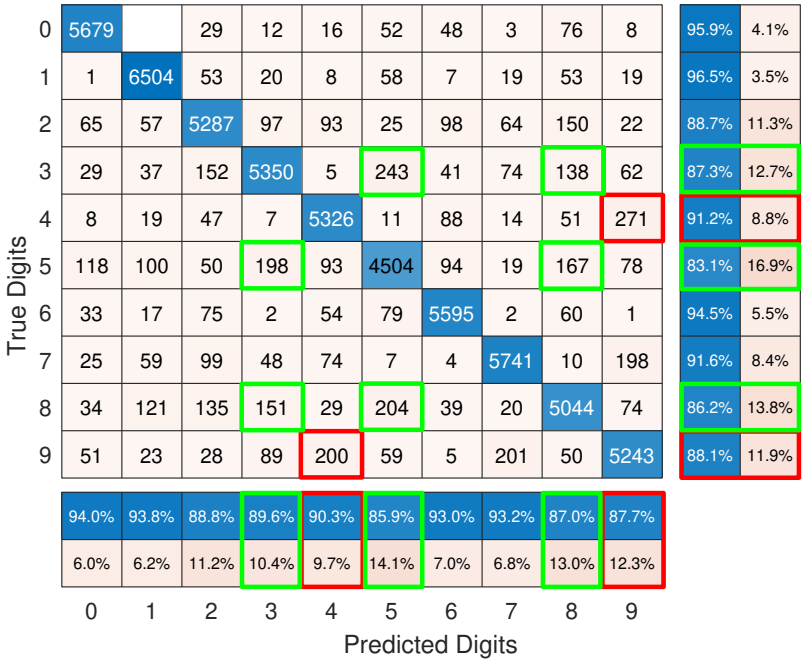

Figure A.17: BBFWA Set 1 PCA MNIST Best Training Trial Confusion Matrix

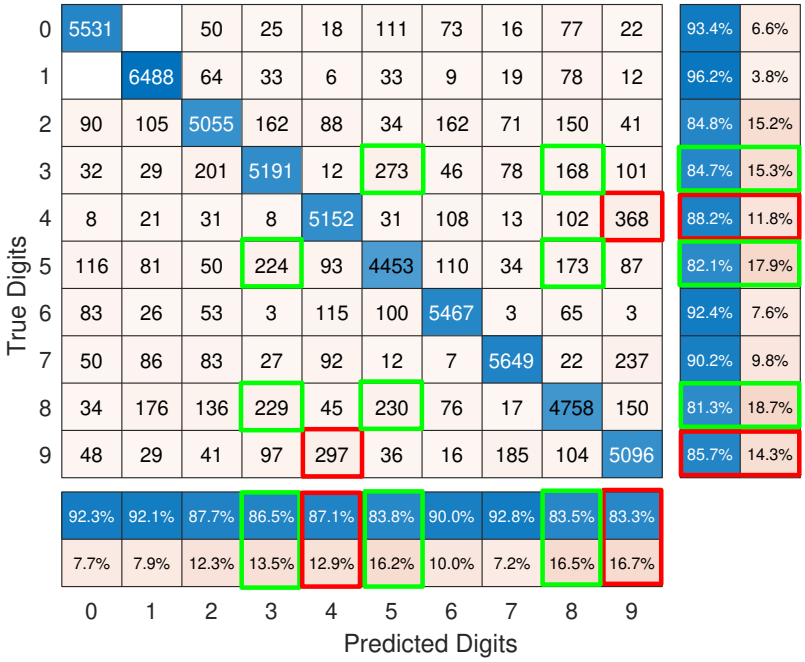

Figure A.18: BBFWA Set 1 PCA MNIST Worst Training Trial Confusion Matrix

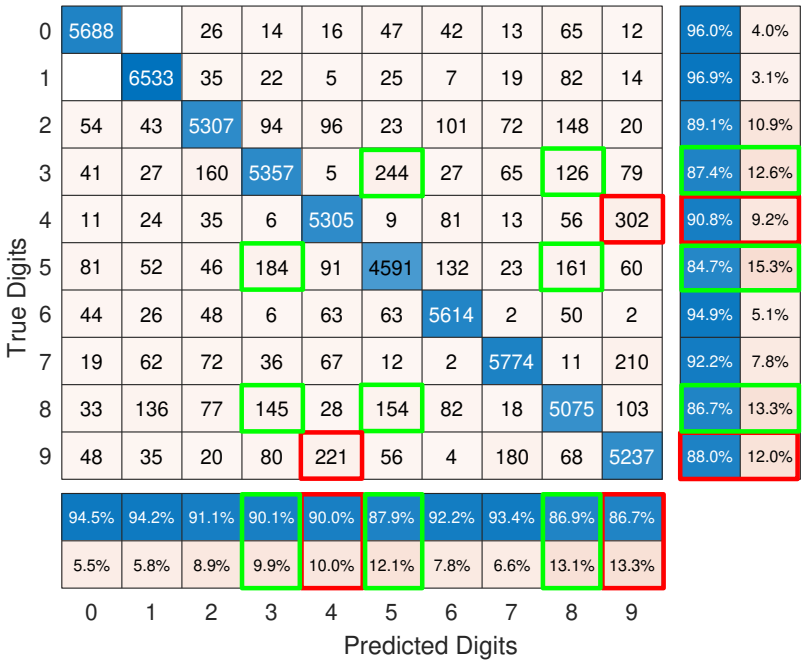

Figure A.19: BBFWA Set 2 PCA MNIST Best Training Trial Confusion Matrix

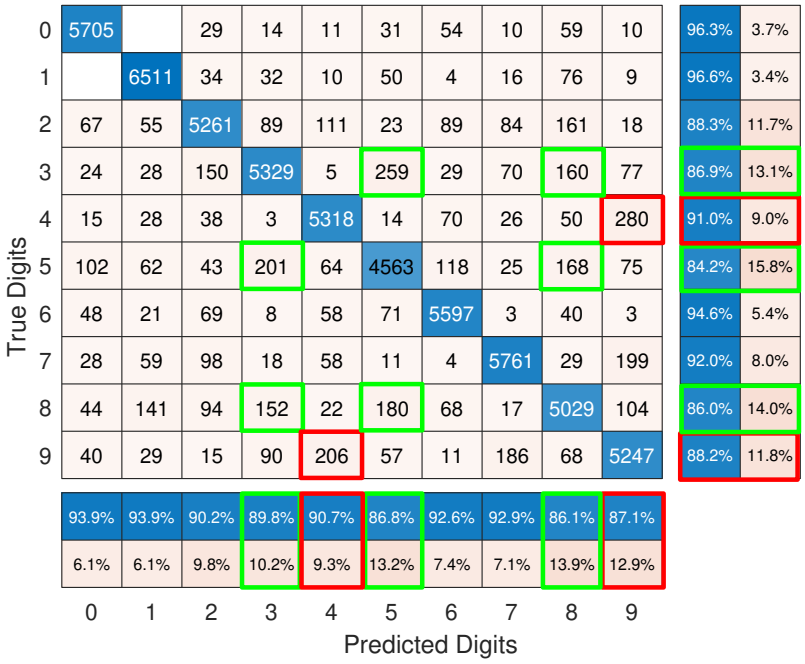

Figure A.20: BBFWA Set 2 PCA MNIST Worst Training Trial Confusion Matrix

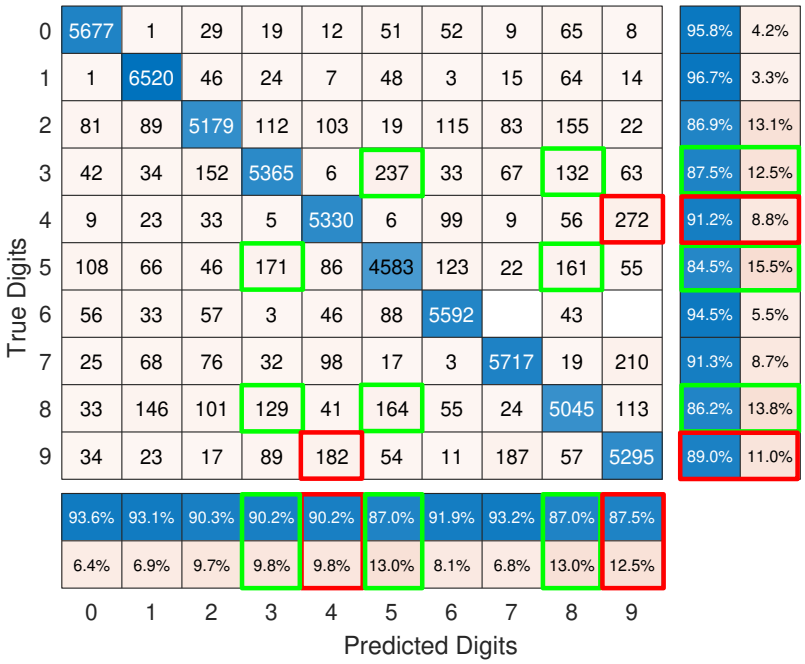

Figure A.21: PSO PCA MNIST Best Training Trial Confusion Matrix

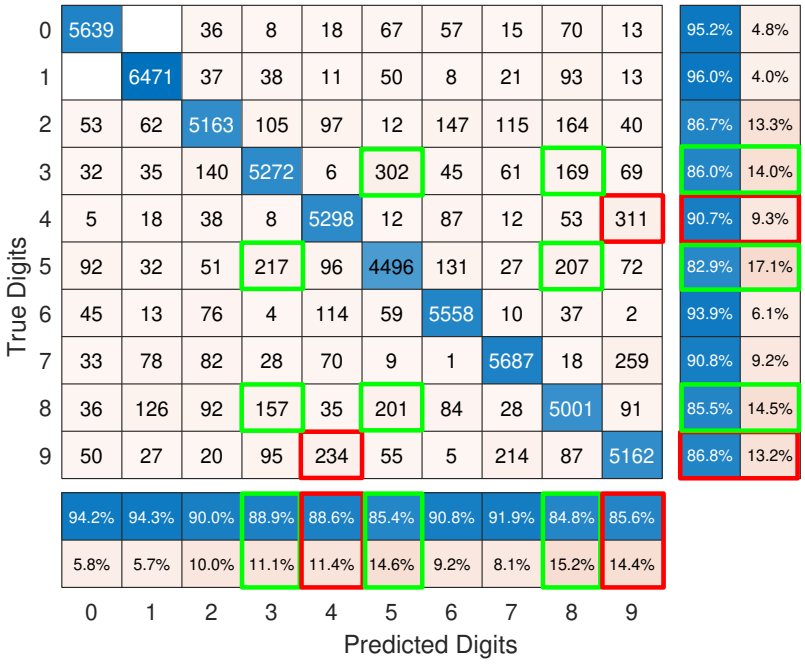

Figure A.22: PSO PCA MNIST Worst Training Trial Confusion Matrix

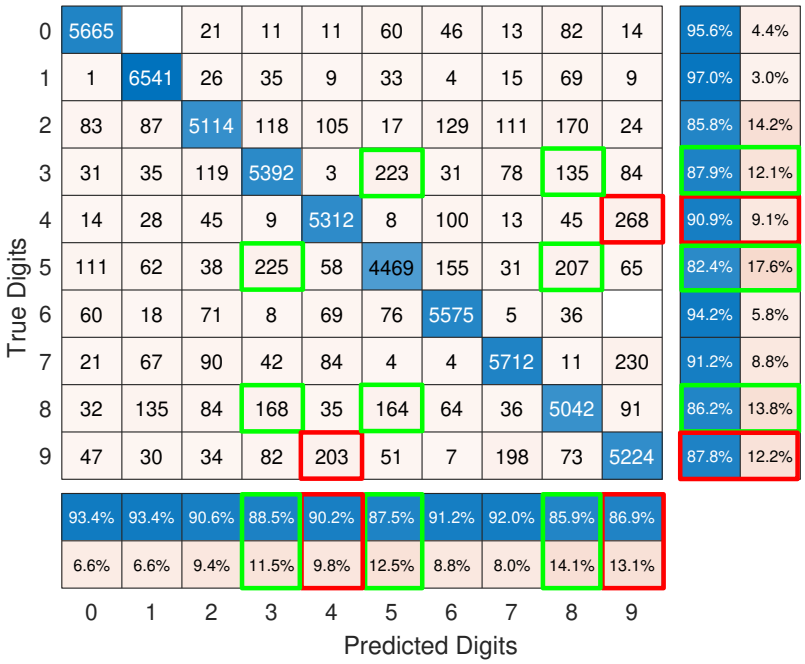

Figure A.23: CPSO PCA MNIST Best Training Trial Confusion Matrix

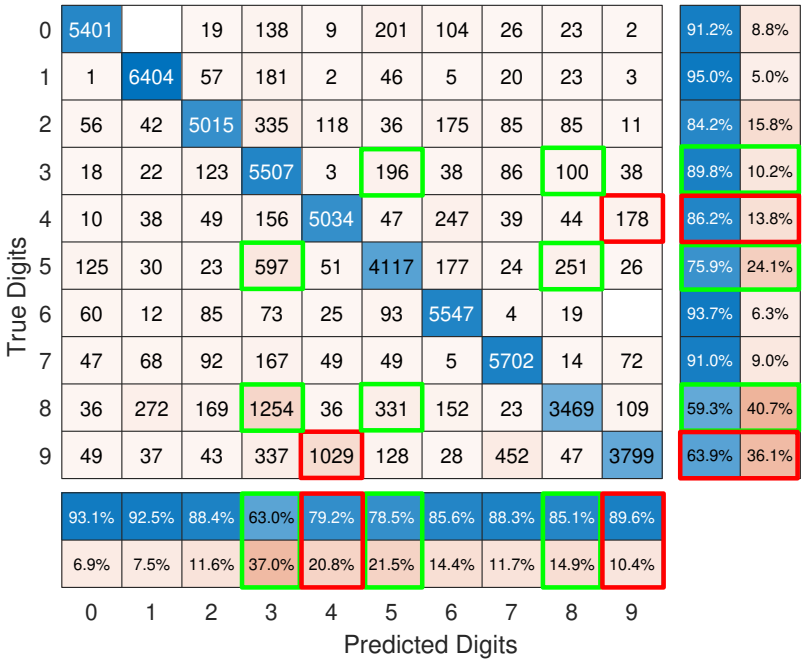

Figure A.24: CPSO PCA MNIST Worst Training Trial Confusion Matrix

APPENDIX B: SOURCE CODE

B.1 Simulation Framework Code

```
% Thesis Final ANN Simulation Framework
% Created by: Patrik Gilley
% Date Created: November 27, 2019
% Date Last Modified: April 15, 2020
% Program name: ANN_Simulation_Frame.m
%
% This program is intended to serve as a simulation framework for testing
% the various ANN training algorithm permutations created as a part of my
% thesis work, and store data and figures from the tests for further
% analysis. Algorithm parameters are stored within object style
% containers to simplify the input arguments of the composite functions.
% This framework will automatically create folders to save the results in.
% It will create a folder for the day it is executed on, and then will
% label individual trial runs with time of execution, the general
% parameters used, and the fitness function used. The final results will
% be stored in a .mat file labelled with key parameters, and the
% individual algorithm files will save their figures to that same folder.
%
% User-defined Functions:
%   sim_predict_final.m: Program to provide selection from list of ...
    algorithm fitness functions.
%   Canon_PSO_ANN_sim.m: Canonical PSO algorithm program.
%   Simulation_Barebones_PSO.m: Barebones PSO algorithm program.
%   Simulation_Canon_FWA.m: Canonical FWA algorithm program.
%   Simulation_Barebones_FWA.m: Barebones FWA algorithm program.
%   draw_ground_truth.m: Used to create function data for plotting of ...
    the ground truth of a fitness function or benchmark function.
%   save_my_figs.m: Application sourced from Dr. Yan to save generated
%                   figures into .fig and .jpeg files.

% Clear Out Previous Program Executions
%*************************************************************************
clear;
close all;

% ANN Parent Data Set Loading/Initialization
%*************************************************************************
dataset_type = 'mnist'; % Controls which data set is used for the
                        % simulation. Options are 'mnist' for using the
                        % MNIST database, and 'ex4' for using an example
                        % data set for verifying/testing programs.
dataset_num = 2;  % Variable that allows for easy switching of data set
                  % used by the framework. dataset_num = 1 uses the
                  % standard, unaltered version of the data set, and
                  % dataset_num = 2 uses the PCA compressed version of the
```

```
                    % data set.

% Data set options for example data use/testing of programs.
if(strcmp(dataset_type,'ex4'))
    % Standard, unaltered parent data set.
    if(dataset_num == 1)
        load('ex4data1.mat');

        % Create a variable to denote what type of database is used. Kept
        % for file naming and figure labelling purposes.
        sub_type = 'standard';

    % PCA dimension reduced parent data set.
    elseif(dataset_num == 2)
        load('ex4data1_pca.mat');

        % Create a variable to denote what type of database is used. Kept
        % for file naming and figure labelling purposes.
        sub_type = 'pca';

        % Remove internal reference variables from the PCA data set. These
        % are still present in the file; this justs keeps them from
        % cluttering local memory during sims.
        clear U pca_comp

        % Matches the data sample array to naming convention used in the
        % program.
        X = X_reduced;
    end
    m = size(X, 1); % Detects the number of samples in the data set.

    % Sets the overall size of the digit classes (number of data samples
    % in the parent data set).
    n_class = 500;

    % Set the percentage of the parent data set samples that will be set
    % aside for the training data set.
    per_train = 0.8;

    % Separate the parent data set randomly into training and testing
    % data sets.
    [X_train,X_test,y_train,y_test,size_train,size_test] = ...
        ann_traintest_single(n_class,per_train,X,y);

% MNIST database data set options.
elseif(strcmp(dataset_type,'mnist'))
    % Standard, unaltered parent data set.
    if(dataset_num == 1)
        load('MNISTdata.mat');
        % Set training and testing data set sample matrices and label
        % matrices.
        X_train = train_images;
        X_test = test_images;
        y_train = train_labels;
```

```
98          y_test = test_labels;
99          X = X_train;
100         y = y_train;
101
102         % Create a variable to denote what type of database is used. Kept
103         % for file naming and figure labelling purposes.
104         sub_type = 'standard';
105
106     % PCA dimension reduced parent data set.
107     elseif(dataset_num == 2)
108         load('MNISTdata_pca.mat');
109         % Remove internal reference variables from the PCA data set.
110         % These are still present in the file; this justs keeps them from
111         % cluttering local memory during sims.
112         clear U pca_comp
113
114         % Set training and testing data set sample matrices and label
115         % matrices.
116         X_train = train_images;
117         X_test = test_images;
118         y_train = train_labels;
119         y_test = test_labels;
120         X = X_train;
121         y = y_train;
122
123         % Create a variable to denote what type of database is used. Kept
124         % for file naming and figure labelling purposes.
125         sub_type = 'pca';
126     end
127 end
128
129 % Simulation Parameter Initialization
130 %*************************************************************************
131 % Set the number of trials the simulation uses.
132 num_trials = 50;
133
134 % Set whether or not the framework should regenerate the BP ANN.
135 % run_BP = 1 will train an ANN with BP for the search algorithms to use,
136 % run_BP = 0 loads a pre-trained ANN coniguration.
137 run_BP = 0;
138
139 % Name of the fitness function. Stored for reference in file naming and
140 % figure labelling.
141 fxn_name = 'Cross_Entropy';
142
143 % General Algorithm Parameter Definition
144 %*************************************************************************
145 param.hid_size = 35;    % Size of the ANN hidden layer.
146 param.num_lab = 10;     % Size of the ANN output layer.
147
148 % Defines solution space bounds for the algorithms.
149 param.uppbound = 100;   % Upper bound.
150 param.lowbound = -100;  % Lower bound.
151
```

```
152 % Max training iterations allowed for BBFWA and PSO algorithms.
153 param.main_maxiter = 300;
154
155 % Max training iterations allowed for CPSO algorithm.
156 param.coop_maxiter = param.main_maxiter/2;
157
158 param.bp_maxiter = 50; % Max training iterations allowed for BP algorithm.
159 param.lambda = 1;% Sets the regularization parameter for the ANN training.
160
161 % Seed value that sets the range in which the nodal weigths will be
162 % initialized.
163 param.epsilon_init = 0.12;
164
165 % Algorithm Internal Plotting Controls
166 %************************************************************************
167 % Set whether the confusion matrix plotting code should be executed.
168 plot_control.will_plot = false;
169
170 % Set whether the program pauses on intermediate confusion matrix plots.
171 plot_control.will_pause = false;
172
173 % Set the interval between intermediate confusion matrix updates.
174 plot_control.iter_interval = 10;
175
176 % Control whether the ANN training programs will generate and save figures.
177 plot_control.final_plot = true;
178
179 % PSO Algorithm Parameter Definition
180 %************************************************************************
181 pso.swarm = 30;     % Number of particles used.
182 pso.c1 = 1.325;     % Individual exploration weight.
183 pso.c2 = 1.325;     % Group exploration weight.
184 pso.omega = 0.7298; % Inertial weight.
185
186 % FWA Algorithm Parameter Definition
187 %************************************************************************
188 fwa.bbspk = 30;    % Sets the number of sparks for the BBFWA program.
189 fwa.cr = 0.5;       % Explosion amplitude reduction coefficient.
190 fwa.ca = 1.2;       % Explosion amplitude amplification coefficient.
191
192 % Train a base ANN with conjugate gradient backpropagation
193 %************************************************************************
194 % The BP ANN is only going to be generated once and then the data will be
195 % used repeatedly; the BP ANN should only be trained once per data set and
196 % ANN structure.
197 if(run_BP == 1)
198     % This option trains the BP ANN. It should only be used to generate
199     % initial data for the different data sets, as well as regenerating
200     % ANNs if ANN structure gets changed.
201
202     % Run the BP ANN training program.
203     [BP_results,BP_weights] = BP_ANN_sim(X_train,X_test,y_train,...
204         y_test,param);
205
```

```
206     % Name and save the BP ANN training results.
207     bp_name = ['bp_' dataset_type '_' sub_type '.mat'];
208     save(bp_name,'BP_results','BP_weights','param','X_train','X_test',...
209         'y_train','y_test');
210 elseif(run_BP == 0)
211     % This option loads a trained BP ANN to seed the search algorithm's
212     % starting locations with. This is generally the option that should be
213     % used.
214     bp_name = ['bp_' dataset_type '_' sub_type '.mat'];
215     load(bp_name);
216 end
217
218 % Data File Saving
219 %*************************************************************************
220 % Folder creation per day. This creates a folder to hold the simulations
221 % run on a single day.
222
223 % Declare a character string for parent directory location for file saving.
224 savepath = ['C:\Users\PWGil\Documents\Masters_Program_Documents\'...
225     'Thesis_Documents\conference_ann_sim'];
226 savedir = ['Simulation_' date];
227 if ¬exist(savedir, 'dir')    % Checks to see if the directory being created
228                              % already exists.
229     mkdir(savepath,savedir)  % Creates a new folder if one does not exist.
230 end
231
232 % Individual simulation folder creation.
233 time = datestr(now);    % Pulls the current time and date for labelling.
234 sim_time = replace(erase(time,date),':','_');% Removes the date and
235                                              % replaces time : with
236                                              % underscores to match file
237                                              % name conventions.
238 sim_time = erase(sim_time,' ');
239 savepath_ind = [savepath '\' savedir];  % Creates a folder path that
240                                         % includes the date folder.
241 subFolder = ['Simulation_Time_' sim_time '_Gen_Param_',...
242     num2str(param.main_maxiter), '_' fxn_name '_Trials_' ...
243     num2str(num_trials) '_Dataset_' dataset_type];
244 if ¬exist(subFolder, 'dir')    % Checks to see if the directory being
245                                % created already exists.
246     mkdir(savepath_ind,subFolder)  % Creates a new folder if one does not
247                                    % exist.
248 end
249
250 % Simulation data directory creation
251 subsubFolder = 'Simulation_Data';
252 savepath_ind = [savepath '\' savedir '\' subFolder];
253 if ¬exist(subFolder, 'dir')    % Checks to see if the directory being
254                                % created already exists.
255     mkdir(savepath_ind,subsubFolder)% Creates a new folder if one does not
256                                     % exist.
257 end
258
259 % File save path. Generate an individual file saving path dependent on
```

```
% previous folder establishments.
filepath = [savepath_ind '\' subsubFolder];

% Function Handle Definition
%*************************************************************************
% Function handle for the Canonical PSO program.
capso = @Canon_PSO_ANN_sim;
% Function handle for the Esplit Cooperative PSO program.
ecopso = @Coop_CPSO_ANN_esplit_sim;
% Function handle for the Barebones FWA program.
bbfwa = @Barebones_FWA_ANN_sim;

% Main Simulation Loop
%*************************************************************************
for sim = 1:num_trials
    % Evaluate the search algorithm programs with rolling variable names.
    eval(['[bbfwa_results_',num2str(sim),'] = bbfwa(X_train,X_test,'...
        'y_train,y_test,BP_results.final_params,param,fwa,'...
        'plot_control,filepath);']);
    eval(['[ecopso_results_',num2str(sim),'] = ecopso(X_train,X_test,'...
        'y_train,y_test,BP_results.final_params,param,pso,'...
        'plot_control,filepath);']);
    eval(['[capso_results_',num2str(sim),'] = capso(X_train,X_test,'...
        'y_train,y_test,BP_results.final_params,param,pso,'...
        'plot_control,filepath);']);
    % Print out the trial number upon completion of a trial, so that
    % simulation progress can be tracked.
    fprintf('Sim number %i',sim)
end

% Naming of .mat results file.
filename = [fxn_name '_Data_set_' dataset_type '_PSO_',...
    num2str(pso.swarm),'_BBFWA_',num2str(fwa.bbspk),'.mat'];
% Create a name and save path for saving the stored results.
matfile = fullfile(filepath,filename);
save(matfile);
```

```
% Simulation Data Post Processor
% Created by: Patrik Gilley
% Date Created: December 17, 2019
% Date Last Modified: April 15, 2020
% Program Name: ANN_Simulation_Post.m
%
% This program is an adapted version of Simulation_Post.m, a program
% written to analyze the results of Monte Carlo simulations from the search
% algorithm testing phase of my thesis. This version cleans up some old
% code and adapts the analysis mechanisms to perform the analysis required
% by the ANN simulations research.
%
% User-Defined Functions Used:
%   save_my_figs.m: Used to save the error bar plots created as part of the
```

```
%                   data analysis.

% Clear Out Previous Data
%*************************************************************************
clear;
close all;
clc;

% Parameter Initialization and Importing of Simulation Results
%*************************************************************************
gen_dir = ['C:\Users\PWGil\Documents\Masters_Program_Documents\'...
    'Thesis_Documents\conference_ann_sim\Simulation_16-Dec-2019'];
sim_dir = ['Simulation_Time_17_37_04_Gen_Param_300_Cross_Entropy_Trials'...
    '_100_Dataset_mnist'];
sim_filename = 'Cross Entropy_Data_set_mnist_PSO_30_BBFWA_30';
sim_results = load([gen_dir '\' sim_dir '\Simulation_Data\'...
    sim_filename '.mat']);

% Program results save directory establishment

% Pulls the file path that was used to save the simulation results to save
% figures and results to the same directory as the simulation results.
savepath = sim_results.filepath;
if(contains(savepath,'Simulation_Data'))   % Defines the parent directory.
    savepath = erase(savepath,'\Simulation_Data');
end

% Creates a separate folder for the post processed data.
savedir = 'Post_Proc_Data';

% Check to see if the directory being created already exists.
if ¬exist(savedir, 'dir')
    mkdir(savepath,savedir)  % Creates a new folder if one does not exist.
end
% Create a save path for the individual files saved from this program.
filepath = [savepath '\' savedir];

% Accuracy matrix initialization
bbfwa_acc_train = zeros(sim_results.num_trials,1);  % Barebones FWA
bbfwa_acc_test = zeros(sim_results.num_trials,1);

capso_acc_train = zeros(sim_results.num_trials,1);  % Canonical PSO
capso_acc_test = zeros(sim_results.num_trials,1);

ecopso_acc_train = zeros(sim_results.num_trials,1); % Esplit CoPSO
ecopso_acc_test = zeros(sim_results.num_trials,1);

bp_acc_train = sim_results.BP_results.train_acc;    % Backpropagation
bp_acc_test = sim_results.BP_results.test_acc;

% Fitness error bar plot matrix initialization
bbfwa_fit = zeros(sim_results.num_trials,sim_results.param.main_maxiter);
capso_fit = zeros(sim_results.num_trials,sim_results.param.main_maxiter);
ecopso_fit = zeros(sim_results.num_trials,sim_results.param.coop_maxiter);
```

```
69 bp_fit = sim_results.BP_results.train_cost;
70
71 % SA Accuracy Analysis
72 %*************************************************************************
73 % Iteratively search all of the results structures.
74 for acc = 1:sim_results.num_trials
75     % ANN Final Accuracy Data Aggregation
76     %*********************************************************************
77     % Finds the mean squared error between the final result and ground
78     % truth for each dimension, then sums up the individual errors for the
79     % overall solution error.
80     bbfwa_acc_train(acc,:) = sim_results.(['bbfwa_results_'...
81         num2str(acc)]).train_acc;
82     bbfwa_acc_test(acc,:) = sim_results.(['bbfwa_results_'...
83         num2str(acc)]).test_acc;
84
85     capso_acc_train(acc,:) = sim_results.(['capso_results_'...
86         num2str(acc)]).train_acc;
87     capso_acc_test(acc,:) = sim_results.(['capso_results_'...
88         num2str(acc)]).test_acc;
89
90     ecopso_acc_train(acc,:) = sim_results.(['ecopso_results_'...
91         num2str(acc)]).train_acc;
92     ecopso_acc_test(acc,:) = sim_results.(['ecopso_results_'...
93         num2str(acc)]).test_acc;
94
95     % Error Bar Plotting Data Collection
96     %*********************************************************************
97     for iter = 1:(sim_results.param.main_maxiter)
98         bbfwa_fit(acc,iter) = sim_results.(['bbfwa_results_'...
99             num2str(acc)]).gbest_fit(1,:,iter);
100        capso_fit(acc,iter) = sim_results.(['capso_results_'...
101            num2str(acc)]).gbest_fit(1,:,iter);
102    end
103
104    % CPSO was split off from the other two algorithms for this step ...
       due to
105    % the difference in training iterations.
106    for iter = 1:(sim_results.param.coop_maxiter)
107        ecopso_fit(acc,iter) = sim_results.(['ecopso_results_'...
108            num2str(acc)]).gbest_fit(1,:,iter);
109    end
110 end
111
112 % Acquire relevant BP ANN statistics from the pre-trained ANN file.
113 bp_results = sim_results.BP_results;
114
115 % ANN Accuracy Mean and Standard Deviation
116 %*************************************************************************
117 % Barebones FWA accuracies
118 avg_bbfwa_acc_train = mean(bbfwa_acc_train);
119 sigma_bbfwa_acc_train = std(bbfwa_acc_train);
120 avg_bbfwa_acc_test = mean(bbfwa_acc_test);
121 sigma_bbfwa_acc_test = std(bbfwa_acc_test);
```

```
122
123 % Canonical PSO accuracies
124 avg_capso_acc_train = mean(capso_acc_train);
125 sigma_capso_acc_train = std(capso_acc_train);
126 avg_capso_acc_test = mean(capso_acc_test);
127 sigma_capso_acc_test = std(capso_acc_test);
128
129 % Esplit Cooperative PSO accuracies
130 avg_ecopso_acc_train = mean(ecopso_acc_train);
131 sigma_ecopso_acc_train = std(ecopso_acc_train);
132 avg_ecopso_acc_test = mean(ecopso_acc_test);
133 sigma_ecopso_acc_test = std(ecopso_acc_test);
134
135 % Error Bar Plot Generation
136 %**************************************************************************
137 % Set the x-axis iteration count.
138 x_main_iter = 0:(sim_results.param.main_maxiter-1);
139 x_coop_iter = 0:(sim_results.param.coop_maxiter-1);
140
141 % BBFWA Plot
142 % Find the average fitness values path for plotting.
143 ave_bbfwa_fit = mean(bbfwa_fit,1);
144 % Determine the 10th and 90th percentile fitness values across all trials
145 % in each iteration.
146 bbfwa_fit_max = prctile(bbfwa_fit,90,1);
147 bbfwa_fit_min = prctile(bbfwa_fit,10,1);
148
149 figure
150
151 hold all
152 errorbar(x_main_iter,ave_bbfwa_fit,bbfwa_fit_min,bbfwa_fit_max,...
153     'vertical','.','MarkerSize',5,'MarkerEdgeColor','red',...
154     'MarkerFaceColor','red')
155 title('BBFWA Simulation Results Fitness Plot with Error Bars')
156 xlabel('Algorithm Iteration Count')
157 ylabel('Fitness Value')
158 temp_fig = gcf;
159 save_my_figs(temp_fig,filepath,'BBFWA_Error_Bars')
160
161 % CPSO Plot
162 % Find the average fitness values path for plotting.
163 ave_capso_fit = mean(capso_fit,1);
164 % Determine the 10th and 90th percentile fitness values across all trials
165 % in each iteration.
166 capso_fit_max = prctile(capso_fit,90,1);
167 capso_fit_min = prctile(capso_fit,10,1);
168
169 figure
170 hold all
171 errorbar(x_main_iter,ave_capso_fit,capso_fit_min,capso_fit_max,...
172     'vertical','.','MarkerSize',5,'MarkerEdgeColor','red',...
173     'MarkerFaceColor','red')
174 title('CPSO Simulation Results Fitness Plot with Error Bars')
175 xlabel('Algorithm Iteration Count')
```

```
ylabel('Fitness Value')
temp_fig = gcf;
save_my_figs(temp_fig,filepath,'CPSO_Error_Bars')

% Esplit Coop PSO Plot
% Find the average fitness values path for plotting.
ave_ecopso_fit = mean(ecopso_fit,1);
% Determine the 10th and 90th percentile fitness values across all trials
% in each iteration.
ecopso_fit_max = prctile(ecopso_fit,90,1);
ecopso_fit_min = prctile(ecopso_fit,10,1);

figure
hold all
errorbar(x_coop_iter,ave_ecopso_fit,ecopso_fit_min,ecopso_fit_max,...
    'vertical','.','MarkerSize',5,'MarkerEdgeColor','red',...
    'MarkerFaceColor','red')
title(['Esplit Cooperative PSO Simulation Results Fitness Plot'...
    'with Error Bars'])
xlabel('Algorithm Iteration Count')
ylabel('Fitness Value')
temp_fig = gcf;
save_my_figs(temp_fig,filepath,'ECOPSO_Error_Bars')

% Saving of Data Post Processing Results
%**************************************************************************
% Pull the date and time string from the simulation framework file saving
% code for further file naming.
sim_time = sim_results.time;
sim_time = replace(sim_time,':','_');
sim_time = replace(sim_time,' ','_at_');
filename = ['Post_Data_for_' sim_results.fxn_name '_Sim_on_'...
    sim_time '_UTC'];

% Deletes the structure containing the source simulation data and the temp
% variable holding the figure handles to reduce file size and saving time.
clear sim_results temp_fig
matfile = fullfile(filepath,filename);
save(matfile)
```

B.2 Backpropagation Algorithm Code

```
function [results,weights] = BP_ANN_sim(X_train,X_test,y_train,...
    y_test,param)

% Backpropagation Artificial Neural Network Training Algorithm Simulation
% Variant
%
% Created by: Patrik Gilley
```

```
8  % Date Created: November 29, 2019
9  % Date Last Modified: November 29, 2019
10 % Program Name: BP_ANN_sim.m
11 %
12 % Program Inputs:
13 %   X_train = Data set sample array that the ANN will be trained with.
14 %   X_test = Data set sample array that the ANN will be tested with.
15 %   y_train = Data set labels that the ANN will be trained with.
16 %   y_test = Data set labels that the ANN will be tested with.
17 %   param = MatLab structure defined in the simulation framework that
18 %           contains the general algorithm settings.
19 %
20 % Program Outputs:
21 %   results = MatLab structure that holds the algorithm output data.
22 %   Properties:
23 %       final_params = Final nodal weight configuration for the trained
24 %                      ANN.
25 %       train_cost = Final training cost/fitness achieved by the training
26 %                    algorithm.
27 %       train_acc = Final training accuracy achieved by the neural network.
28 %       train_pred = Column vector archiving the final training phase
29 %                    output labels.
30 %       train_time = Total training time of the BP algorithm.
31 %       test_acc = Final testing accuracy achieved by the neural network.
32 %       test_pred = Column vector archiving the testing phase output
33 %                   labels.
34 %       test_time = Total testing time of the BP algorithm.
35 %   weights = MatLab structure containing the final weight matrices of the
36 %             trained ANN.
37 %   Properties:
38 %       theta1 = Nodal weight matrix for the input to hidden layers.
39 %       theta2 = Nodal weight matrix for the hidden to output layers.
40 %
41 % This program is a simulation subprogram for the Backpropagation
42 % algorithm, which is used as a control in the ANN test simulations. The
43 % program takes a training data set and ANN structure and trains an ANN
44 % using all of them. The program evaluates the performance of the
45 % trained ANN on both the training and testing data sets. Both the ANN
46 % performance evaluation and the trained ANN weights are passed out of the
47 % program as outputs for use with other programs.
48
49 % Parameter Initialization
50 %*************************************************************************
51 % Randomly shuffle the data set.
52 size_train = size(X_train,1);
53 shuffle_train = randperm(size_train,size_train);
54 X_train(:,:) = X_train(shuffle_train,:);
55 y_train(:,1) = y_train(shuffle_train,:);
56
57 size_test = size(X_test,1);
58 shuffle_test = randperm(size_test,size_test);
59 X_test(:,:) = X_test(shuffle_test,:);
60 y_test(:,:) = y_test(shuffle_test,:);
61
```

```
% Internal Parameter Initialization
input_layer_size = size(X_train,2);
hidden_layer_size = param.hid_size;
num_labels = param.num_lab;

% ANN Training Phase
%************************************************************************
tic;
% ANN nodal weight matrix initialization
initial_Theta1 = randInitializeWeights(input_layer_size,...
    hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);

% Unroll initialized weight matrices into a full vector for use with
% training program.
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];

% Backpropagation training algorithm definitions.
lambda = param.lambda;  % Reads in framework set regularization constant.

% Define the max number of training iterations allowed for the training
% program.
options = optimset('MaxIter', param.bp_maxiter);

% Cost function handle for the training program to train with.
costFunction = @(p) sim_predict_BP(p, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, X_train, y_train, lambda);
% Call the training program.
[results.final_params, results.train_cost] = fmincg(costFunction,...
    initial_nn_params, options);

% Obtain Theta1 and Theta2 back from nn_params
weights.theta1 = reshape(results.final_params(1:hidden_layer_size...
    * (input_layer_size + 1)),hidden_layer_size, (input_layer_size + 1));

weights.theta2 = reshape(results.final_params((1 + (hidden_layer_size...
    * (input_layer_size + 1))):end),num_labels, (hidden_layer_size + 1));

% Use the trained weights to predict the labels of the training data.
results.train_pred = predict(weights.theta1, weights.theta2, X_train);

% Calculate the overall classification accuracy of the ANN.
results.train_acc = mean(double(results.train_pred == y_train)) * 100;
results.train_time = toc; % Archives the overall training time of the ANN.

% ANN Testing Phase
%************************************************************************
% Repeats post-training use of ANN done in training phase, but with
% testing data set.
results.test_pred = predict(weights.theta1, weights.theta2, X_test);
results.test_acc = mean(double(results.test_pred == y_test)) * 100;
results.test_time = toc;
```

```
function [fit, grad] = sim_predict_BP(nn_weights, input_layer_size,...
    hidden_layer_size, num_labels, X, y, lambda)

% Cost Function Calculator for OCR MLP ANN
% Created by: Patrik Gilley
% Date Created: November 29, 2019
% Date Last Modified: November 29, 2019
% Program Name: sim_predict_BP.m
%
% This program was created to take the trained weights of an ANN and use
% them to predict the label of the data inputs to that ANN. Given a
% database of handwritten digits and the nodal weights of an ANN trained
% for classifying those digits, this program would predict the output label
% of each input image. Currently, this program is set for a specified ANN
% layer structure of one input layer, one hidden layer, and one output
% layer. This is a variant of sim_predict_final.m, created to better match
% the expected syntax of the cost function used in the fmincg.m
% backpropagation training algorithm.
%
% Program Inputs:
%   nn_weights = Unrolled vector of neural network nodal weights. Will be
%                reshaped into weight matrices.
%   input_layer_size = Number of nodes in the input layer of the ANN.
%   hidden_layer_size = Number of nodes in the hidden layer of the ANN.
%   num_labels = Number of output nodes in the ANN.
%   X = The data inputs to the neural network.
%   y = The labels that correspond to the data inputs used with the ANN.
%   lambda = The lambda or regularization constant that the weight
%            regularization calculation should use.
%
% Program Outputs:
%   fit = Overall fitness value of the ANN nodal weights, based on training
%         accuracy.
%   grad = Unrolled vector of calculated gradients.
%
% NOTE: The code partially adapts code from a Coursera online course about
%       machine learning created by Dr. Andrew Ng.

% Feed Training Data Through ANN
%*********************************************************************
m = size(X, 1);   % Identifies the number of input data entries.
Theta1 = reshape(nn_weights(1:hidden_layer_size*(input_layer_size + 1)),...
    hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_weights((1+(hidden_layer_size*(input_layer_size+1))))...
    :end),num_labels, (hidden_layer_size + 1));

% Feed input data through activation functions.
h1 = relu_sim([ones(m, 1) X] * Theta1');
```

```
50 h2 = softmax_sim([ones(m, 1) h1] * Theta2');
51
52 % Correction for number rounding.
53 %************************************************************************
54 % Makes sure that there are no actual ones in the final h2 output matrix.
55 % Subtracts a very small number from found 1's in order to make them
56 % slightly less than one without overtly affecting the results of the
57 % neural network.
58 h2(h2==1) = h2(h2==1) - 1e-15;
59
60 % Fitness/Cost Calculation
61 %*****************************************************************
62 % Convert data set labels into ANN output layer format.
63 I = eye(10);
64 Y = zeros(m, 10);
65 for i=1:m
66     Y(i, :)= I(y(i), :);
67 end
68
69 % Regularization
70 % Remove unregularized bias nodal weights from weights to be regularized.
71 reg_theta1 = Theta1(:,2:size(Theta1,2));
72 reg_theta2 = Theta2(:,2:size(Theta2,2));
73 % Calculate regularization term of cost function.
74 reg = (lambda/(2*m))*(sum(sum((reg_theta1.^2),2))+...
75     sum(sum((reg_theta2.^2),2)));
76
77 % Calculate the cost function based on outputss and converted data set
78 % labels.
79 fit = (1/m)*sum(sum((-Y.*log(h2))-(1-Y).*log(1-h2),2))+reg;
80
81 % Backpropagation Gradient Calculations
82 %*************************************************************
83 % Parameter Initialization
84 Delta1 = zeros(size(Theta1));
85 Delta2 = zeros(size(Theta2));
86
87 for t = 1:m
88     % Hidden layer activation function output calculation
89     a_1 = [ones(1, 1) X(t,:)];
90     z_2 = a_1 * Theta1';
91     a_2 = relu_sim(z_2);
92
93     % Output layer activation outputs
94     % Add a bias term to the activation matrix.
95     alt_a_2 = ones(size(a_2,1),(size(a_2,2)+1));
96     alt_a_2(:,2:(size(a_2,2)+1)) = a_2;
97     z_3 = alt_a_2 * Theta2';
98     a_3 = softmax_sim(z_3);
99
100    % Delta error calcs
101    % Calculate the errors of the output layer.
102    d_3 = (a_3 - Y(t,:))';
103    % Calculate the errors of the hidden layer.
```

```
    Δ_2 = (Theta2(:,2:size(Theta2,2))'*Δ_3)'.*reluGradient(z_2);

    % Accumulated errors
    Delta2 = Delta2 + Δ_3*(alt_a_2);
    Delta1 = Delta1 + Δ_2'*(a_1);
end

% Regularization
% Copy the theta matrices.
reg_Theta1 = Theta1;
reg_Theta2 = Theta2;
%j≠0
reg_Theta1(:,1:size(Theta1,2)) = (lambda/m)*Theta1;
reg_Theta2(:,1:size(Theta2,2)) = (lambda/m)*Theta2;
%j=0
reg_Theta1(:,1) = 0;
reg_Theta2(:,1) = 0;

% Unregularized gradient calculations using partial derivative formula
% (1/m)*Δ
Theta1_grad = (1/m)*Delta1+reg_Theta1;
Theta2_grad = (1/m)*Delta2+reg_Theta2;

% Unroll gradients
grad = [Theta1_grad(:) ; Theta2_grad(:)];
end
```

```
function [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
% Minimize a continuous differentialble multivariate function. Starting
% point is given by "X" (D by 1), and the function named in the string "f",
% must return a function value and a vector of partial derivatives. The
% Polack-Ribiere flavour of conjugate gradients is used to compute search
% directions, and a line search using quadratic and cubic polynomial
% approximations and the Wolfe-Powell stopping criteria is used together
% with the slope ratio method for guessing initial step sizes. Additionally
% a bunch of checks are made to make sure that exploration is taking place
% and that extrapolation will not be unboundedly large. The "length" gives
% the length of the run: if it is positive, it gives the maximum number of
% line searches, if negative its absolute gives the maximum allowed number
% of function evaluations. You can (optionally) give "length" a second
% component, which will indicate the reduction in function value to be
% expected in the first line-search (defaults to 1.0). The function returns
% when either its length is up, or if no further progress can be made (ie,
% we are at a minimum, or so close that due to numerical problems, we
% cannot get any closer). If the function terminates within a few
% iterations, it could be an indication that the function value and
% derivatives are not consistent (ie, there may be a bug in the
% implementation of your "f" function). The function returns the found
% solution "X", a vector of function values "fX" indicating the progress
% made and "i" the number of iterations (line searches or function
% evaluations, depending on the sign of "length") used.
```

```
%
% Usage: [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
%
% See also: checkgrad
%
% Copyright (C) 2001 and 2002 by Carl Edward Rasmussen. Date 2002-02-13
%
%
% (C) Copyright 1999, 2000 & 2001, Carl Edward Rasmussen
%
% Permission is granted for anyone to copy, use, or modify these
% programs and accompanying documents for purposes of research or
% education, provided this copyright notice is retained, and note is
% made of any changes that have been made.
%
% These programs and documents are distributed without any warranty,
% express or implied.  As the programs were written for research
% purposes only, they have not been tested to the degree that would be
% advisable in any important application.  All use of these programs is
% entirely at the user's own risk.
%
% [ml-class] Changes Made:
% 1) Function name and argument specifications
% 2) Output display
%

% Read options
if exist('options', 'var')&&¬isempty(options)&&isfield(options, 'MaxIter')
    length = options.MaxIter;
else
    length = 100;
end


RHO = 0.01;  % a bunch of constants for line searches
SIG = 0.5;   % RHO and SIG are the constants in the Wolfe-Powell conditions
INT = 0.1;% don't reevaluate within 0.1 of the limit of the current bracket
EXT = 3.0;% extrapolate maximum 3 times the current bracket
MAX = 20;    % max 20 function evaluations per line search
RATIO = 100; % maximum allowed slope ratio

argstr = ['feval(f, X'];            % compose string used to call function
for i = 1:(nargin - 3)
  argstr = [argstr, ',P', int2str(i)];
end
argstr = [argstr, ')'];

if max(size(length)) == 2, red=length(2); length=length(1); else red=1; end
S=['Iteration '];

i = 0;                                % zero the run length counter
ls_failed = 0;                         % no previous line search has failed
fX = [];
[f1 df1] = eval(argstr);              % get function value and gradient
```

```
i = i + (length<0);                      % count epochs?!
s = -df1;                               % search direction is steepest
d1 = -s'*s;                              % this is the slope
z1 = red/(1-d1);                        % initial step is red/(|s|+1)

while i < abs(length)                   % while not finished
  i = i + (length>0);                     % count iterations?!

  X0 = X; f0 = f1; df0 = df1;           % make a copy of current values
  X = X + z1*s;                           % begin line search
  [f2 df2] = eval(argstr);
  i = i + (length<0);                   % count epochs?!
  d2 = df2'*s;
  f3 = f1; d3 = d1; z3 = -z1;     % initialize point 3 equal to point 1
  if length>0, M = MAX; else M = min(MAX, -length-i); end
  success = 0; limit = -1;                     % initialize quanteties
  while 1
    while ((f2 > f1+z1*RHO*d1) || (d2 > -SIG*d1)) && (M > 0)
      limit = z1;                                     % tighten the bracket
      if f2 > f1
        z2 = z3 - (0.5*d3*z3*z3)/(d3*z3+f2-f3);           % quadratic fit
      else
        A = 6*(f2-f3)/z3+3*(d2+d3);                              % cubic fit
        B = 3*(f3-f2)-z3*(d3+2*d2);
        z2 = (sqrt(B*B-A*d2*z3*z3)-B)/A; % numerical error possible - ok!
      end
      if isnan(z2) || isinf(z2)
        z2 = z3/2;            % if we had a numerical problem then bisect
      end
      z2 = max(min(z2, INT*z3),(1-INT)*z3);  % don't accept too close
                                             % to limits
      z1 = z1 + z2;                                       % update the step
      X = X + z2*s;
      [f2 df2] = eval(argstr);
      M = M - 1; i = i + (length<0);                       % count epochs?!
      d2 = df2'*s;
      z3 = z3-z2;              % z3 is now relative to the location of z2
    end
    if f2 > f1+z1*RHO*d1 || d2 > -SIG*d1
      break;                                            % this is a failure
    elseif d2 > SIG*d1
      success = 1; break;                                          % success
    elseif M == 0
      break;                                                       % failure
    end
    A = 6*(f2-f3)/z3+3*(d2+d3);                % make cubic extrapolation
    B = 3*(f3-f2)-z3*(d3+2*d2);
    z2 = -d2*z3*z3/(B+sqrt(B*B-A*d2*z3*z3));   % num. error possible - ok!
    if ¬isreal(z2) || isnan(z2) || isinf(z2) || z2 < 0 % num prob or wrong
                                                       % sign?
      if limit < -0.5                               % if we have no upper limit
        z2 = z1 * (EXT-1);            % the extrapolate the maximum amount
      else
        z2 = (limit-z1)/2;                                % otherwise bisect
```

```
      end
    elseif (limit > -0.5) && (z2+z1 > limit)    % extraplation beyond max?
      z2 = (limit-z1)/2;                                        % bisect
    elseif (limit < -0.5) && (z2+z1 > z1*EXT) % extrapolation beyond limit
      z2 = z1*(EXT-1.0);                      % set to extrapolation limit
    elseif z2 < -z3*INT
      z2 = -z3*INT;
    elseif (limit > -0.5) && (z2 < (limit-z1)*(1.0-INT))% too close to
                                                        % limit?
      z2 = (limit-z1)*(1.0-INT);
    end
    f3 = f2; d3 = d2; z3 = -z2;          % set point 3 equal to point 2
    z1 = z1 + z2; X = X + z2*s;                % update current estimates
    [f2 df2] = eval(argstr);
    M = M - 1; i = i + (length<0);                          % count epochs?!
    d2 = df2'*s;
  end                                                  % end of line search

  if success                                          % if line search succeeded
    f1 = f2; fX = [fX' f1]';
    fprintf('%s %4i | Cost: %4.6e\r', S, i, f1);
    s = (df2'*df2-df1'*df2)/(df1'*df1)*s - df2; % Polack-Ribiere direction
    tmp = df1; df1 = df2; df2 = tmp;                    % swap derivatives
    d2 = df1'*s;
    if d2 > 0                                      % new slope must be negative
      s = -df1;                           % otherwise use steepest direction
      d2 = -s'*s;
    end
    z1 = z1 * min(RATIO, d1/(d2-realmin));     % slope ratio but max RATIO
    d1 = d2;
    ls_failed = 0;                              % this line search did not fail
  else
    X = X0; f1 = f0; df1 = df0;      % restore point from before
                                     % failed line search
    if ls_failed || i > abs(length)    % line search failed twice in a row
      break;                         % or we ran out of time, so we give up
    end
    tmp = df1; df1 = df2; df2 = tmp;                    % swap derivatives
    s = -df1;                                               % try steepest
    d1 = -s'*s;
    z1 = 1/(1-d1);
    ls_failed = 1;                                % this line search failed
  end
  if exist('OCTAVE_VERSION')
    fflush(stdout);
  end
end
fprintf('\n');
```

```
function [grad] = reluGradient(in)
% Rectified Linear Unit Gradient/Derivative Calculator
```

```
% Created by: Patrik Gilley
% Date Created: December 5, 2019
% Date Last Modified: December 5, 2019
% Program Name: reluGradient.m
%
% Program Inputs:
%   in = Nodal weight matrix to be transformed. Must have already had any
%        relevant transformations (Data samples, previous error terms,
%        etc.) applied to it.
%
% Program Outputs:
%   grad = ReLU gradient of the input weights. Returns with the same
%          dimensions of the input.
%
%
% Computes the gradient of the ReLU function. The derivative of relu is
% f'(x) = 1 if x>0,0 if x<0, && undefined/Inf if x=0. The undefined case
% is set to zero for simplicity, as well as keeping the gradient from
% blowing up on the off chance of a zero result.

% Gradient Calculation
%*************************************************************************
grad = ones(size(in)).*(in>0);
end
```

```
function W = randInitializeWeights(L_in, L_out)
%RANDINITIALIZEWEIGHTS Randomly initialize the weights of a layer with L_in
%incoming connections and L_out outgoing connections
%   W = RANDINITIALIZEWEIGHTS(L_in, L_out) randomly initializes the weights
%   of a layer with L_in incoming connections and L_out outgoing
%   connections.
epsilon_init = 0.12;
W = rand(L_out,1 + L_in)*2*epsilon_init - epsilon_init;
end
```

```
function p = predict(Theta1, Theta2, X)
%PREDICT Predict the label of an input given a trained neural network
%   p = PREDICT(Theta1, Theta2, X) outputs the predicted label of X ...
    given the
%   trained weights of a neural network (Theta1, Theta2)
m = size(X, 1);
h1 = relu_sim([ones(m, 1) X] * Theta1');
h2 = softmax_sim([ones(m, 1) h1] * Theta2');
[¬, p] = max(h2, [], 2);
end
```

B.3 Search Algorithm Code

B.3.1 Bare Bones Fireworks Algorithm

```
function [results]=Barebones_FWA_ANN_sim(X_train,X_test,y_train,y_test,...
    seed_nnparam,param,fwa,plot_control,filepath)

% Barebones Firework Algorithm Artificial Neural Network Training Algorithm
% Simulation Variant
%
% Created By: Patrik Gilley
% Program name: Barebones_FWA_ANN_sim.m
% Date Created: November 29, 2019
% Date Last Modified: December 13, 2019
%
% This program is the final version of the Barebones FWA ANN training
% program, reformatted for use in a Monte Carlo simulation framework. This
% is the master program for the BBFWA ANN training algorithm. All of the
% neural network programs and functions are used within this program and
% its subroutines/functions. The only program on a higher level than this
% program is the simulation script itself.
%
% Program Inputs:
%   X_train = Data set sample array that the ANN will be trained with.
%   X_test = Data set sample array that the ANN will be tested with.
%   y_train = Data set labels that the ANN will be trained with.
%   y_test = Data set labels that the ANN will be tested with.
%   seed_nnparam = Pre-trained ANN weights that the ANN that will be
%                  trained by this program should be seeded with.
%   param = MatLab structure defined in the simulation framework that
%           contains the general algorithm settings.
%   fwa = MatLab structure defined in simulation framework that contains
%         FWA algorithm specific settings.
%   plot_control = MatLab structure defined in simulation framework that
%                  contains confusion matrix plotting settings.
%   filepath = String path to directory that figures will be saved to.
%
% Program Outputs:
%   results = MatLab structure that holds the algorithm output data.
%   Properties:
%       gbest_fit = The array of best global fitness values achieved by the
%                   firework.
%       gbest_pos = Vector containing the best position achieved by the
%                   firework.
%       Training Metrics:
%           train_time = Total training time of the BBFWA algorithm.
%           train_acc = Final training accuracy achieved by the neural
%                       network.
%           train_fit = Final training fitness achieved by the training
%                       algorithm.
%           train_pred = Column vector archiving the final training phase
%                        output labels.
```

```
49 %           train_confuse=Confusion matrix for the gbest ANN predictions
50 %                         on the final evaluation of the training data set.
51 %       Testing Metrics:
52 %           test_time = Total testing time of the BBFWA algorithm.
53 %           test_acc = ANN classification accuracy on the testing data set.
54 %           test_fit = BBFWA fitness from the testing phase evaluation.
55 %           test_pred = Column vector archiving the output labels from the
56 %                       test phase evaluation.
57 %           test_confuse = Confusion matrix for the gbest ANN predictions
58 %                          on the testing data set.
59 %
60 % User Defined Functions:
61 %   sim_predict_final.m: ANN fitness function evaluator for the search
62 %                        algorithms.
63 %   confusemat_plot.m: Creates confusion matrix plots to display the
64 %                      progress of the ANN training.
65
66 % Clear Out Previous Figures
67 %*************************************************************************
68 close all;
69
70 % Neural Network Parameter Initialization
71 %*************************************************************************
72 % Randomly shuffle the data set.
73 size_train = size(X_train,1);   % Determines number of samples in data set.
74 % Create an integer vector from 1 to size_train with randomly sorted
75 % values.
76 shuffle_train = randperm(size_train,size_train);
77 % Use random integers as new sample indices.
78 y_train(:,1) = y_train(shuffle_train,:);
79
80 size_test = size(X_test,1);
81 shuffle_test = randperm(size_test,size_test);
82 X_test(:,:) = X_test(shuffle_test,:);
83 y_test(:,:) = y_test(shuffle_test,:);
84
85 % ANN network structure
86 input_layer_size  = size(X_train,2);  % Sets input layer size to number of
87                                       % data features in X_train.
88 hidden_layer_size = param.hid_size;
89 num_labels = param.num_lab;           % 10 labels, from 1 to 10.
90                                       % Label 10 represents 0.
91
92 % Solution Space and Initial Parameter Initialization
93 %*************************************************************************
94 % Control Structure Parameter Setting
95 upper_bounds = param.uppbound;% Sets the upper bound of the solution space.
96 lower_bounds = param.lowbound;% Sets the lower bound of the solution space.
97 max_iter = param.main_maxiter;% Maximum number of iterations allowed.
98 epsilon_init = param.epsilon_init;  % Seed value that sets the range in
99                                     % which the nodal weigths will be
100                                    % initialized.
101 lambda = param.lambda;  % Regularization constant.
102 iter_interval = plot_control.iter_interval;     % Sets the interval between
```

```
                                                    % intermediate confusion
                                                    % matrix updates.
num_spark = fwa.bbspk;  % Sets number of sparks per iteration.
c_r = fwa.cr;  % Sets the reduction coefficient to tighten the explosion
               % radius if the best solution is within the search range.

c_a = fwa.ca;  % Sets the amplification coefficient to widen the
               % explosion radius if the best solution is not within the
               % search range.

% Internal Parameter Initialization
% Find the size of the nodal weight matrices.
theta1_size = hidden_layer_size*(input_layer_size+1);
theta2_size = num_labels*(hidden_layer_size+1);
dim_num = theta1_size+theta2_size;    % Uses the ANN structure to determine
                                      % the number of weights that need to
                                      % be set. Accounts for bias nodes.
exp_amp = (upper_bounds-(lower_bounds))/2;% Finds the radius of the initial
                                          % firework explosion amplitude
                                          % based on the solution space
                                          % dimensions.
iter = 1;   % Initializes an iteration tracking variable.
dummy = 0;  % Creates a dummy variable for confusemat_plot.m figure ...
   handle input.

% Firework metric matrix initialization
confuse_mat = zeros(num_labels,num_labels,num_spark); % Single iteration
                                                      % storage of
                                                      % confusion matrices.
ANN_pred = zeros(size(X_train,1),num_spark);    % ANN output labels.
ANN_acc = zeros(1+num_spark,max_iter);     % ANN classification accuracies.
fire_pos = zeros(1,dim_num,max_iter);      % Firework positions.

% Spark positions. Row per spark and column per dimension.
spark_pos = zeros(num_spark,dim_num,max_iter);

% Spark fitness values. Initialized to infinity to prevent initial values
% from interfereing with fitness check.
spark_fit = ones(num_spark,1,max_iter)*Inf;
exp_amp_hist = zeros(1,max_iter);   % Explosion amplitude storage.

% Firework Initialization
%**************************************************************************
tic
pos_sign = (randi([0,1],1,dim_num)*2)-1;  % Randomly determine the sign of
                                          % the matrix position coordinate.

% Seeding code for giving the algorithm a pre-trained ANN by giving an
% unrolled vector of nodal weights for the entire network.
seed_pos = seed_nnparam';   % Copies the pre-trained ANN weights.

% Implement input to hidden layer weights with symmetry breaking
% random numbers.
fire_pos(:,1:theta1_size,1) = seed_pos(:,1:theta1_size)+...
```

```
156     pos_sign(:,1:theta1_size).*(epsilon_init*2*rand(1,theta1_size)...
157     -epsilon_init);
158
159 % Implement hidden to output layer weights with symmetry breaking
160 % random numbers.
161 fire_pos(:,(theta1_size+1):dim_num,1)=seed_pos(:,...
162     (theta1_size+1):dim_num)+pos_sign(:,(theta1_size+1):dim_num).*...
163     (epsilon_init*2*rand(1,(dim_num-theta1_size))-epsilon_init);
164
165 % Acquire and reshape the data for the Theta matrix parameters
166 % to fit ANN structure.
167 Theta1 = reshape(fire_pos(1,1:theta1_size),...
168     [hidden_layer_size input_layer_size+1]);
169 Theta2 = reshape(fire_pos(1,(theta1_size+1):dim_num),...
170     [num_labels hidden_layer_size+1]);
171
172 % Determine the fitness of the firework.
173 [fire_fit(1,:,1),ANN_acc(1,1),ANN_pred(:,1),confuse_mat(:,:,1)] = ...
174     sim_predict_final(Theta1,Theta2,X_train,y_train,lambda);
175
176 % If plots are enabled, a confusion matrix will be plotted for the global
177 % best particle solution.
178 if(plot_control.will_plot)
179    sol_select = 1;
180    ann_fit = sol_select;
181    [inter_fig]=confusemat_plot(confuse_mat(:,:,1), sol_select, iter,...
182        ann_fit, max_iter, iter_interval, dummy, plot_control.will_pause);
183 end
184
185 % Firework Algorithm Main Execution
186 %*************************************************************************
187 while(iter < max_iter)
188     % Spark Generation and Fitness Evaluation
189     %*********************************************************************
190     % Randomly determine matrix position coordinate signs.
191     pos_sign = (randi([0,1],num_spark,dim_num)*2)-1;
192
193     % Randomize the spark locations within the explosion zone.
194     spark_pos(:,:,iter) = fire_pos(1,:,iter)+...
195         (exp_amp.*rand(num_spark,dim_num).*pos_sign);
196     exp_amp_hist(1,iter) = exp_amp; % Archive current explosion amplitude.
197
198     % Reshape spark positions into ANN weight matrices.
199     Theta1 = reshape(spark_pos(:,1:theta1_size,iter)',...
200         [hidden_layer_size input_layer_size+1 num_spark]);
201     Theta2 = reshape(spark_pos(:,(theta1_size+1):dim_num,iter)',...
202         [num_labels hidden_layer_size+1 num_spark]);
203
204     % Evaluate the fitness of the sparks.
205     for ann_fit = 1:num_spark
206         [spark_fit(ann_fit,:,1),ANN_acc(ann_fit+1,iter),...
207             ANN_pred(:,ann_fit),confuse_mat(:,:,ann_fit)] = ...
208             sim_predict_final(Theta1(:,:,ann_fit),Theta2(:,:,ann_fit),...
209             X_train,y_train,lambda);
```

```
210     end
211
212     % Determine minimum fitness values for recalibration of firework.
213     [best_spark_fit, best_ind] = min(spark_fit(:));
214     % Convert the index contained in best_ind into row and column indices.
215     [row, ¬] = ind2sub(size(spark_fit),best_ind);
216
217     % Check if best_spark_fit is a better solution than the current
218     % firework position.
219     if(best_spark_fit < fire_fit(1,1,iter))
220         % Move the firework to the new best solution.
221         fire_pos(1,:,iter+1) = spark_pos(row,:,iter);
222         % Set firework fitness to the new global best fitness, and set the
223         % ANN accuracy of the spark to the overall ANN accuracy.
224         fire_fit(1,:,iter+1) = best_spark_fit;
225         ANN_acc(1,iter+1) = ANN_acc(row,iter);
226         % Expand the search zone for better exploration.
227         exp_amp = c_a * exp_amp;
228     else
229         % Preserve the current firework solution if better solution
230         % not found.
231         fire_pos(1,:,iter+1) = fire_pos(1,:,iter);
232         fire_fit(1,:,iter+1) = fire_fit(1,:,iter);
233         ANN_acc(1,iter+1) = ANN_acc(1,iter);
234         % Shrink the search zone for better explotation of the search area.
235         exp_amp = c_r * exp_amp;
236     end
237
238     % If plots are enabled, a confusion matrix will be plotted for the
239     % current global best particle solution.
240     if(plot_control.will_plot)
241         sol_select = row;
242         ann_fit = sol_select;
243         [inter_fig] = confusemat_plot(confuse_mat(:,:,ann_fit),...
244             sol_select, iter, ann_fit, max_iter, iter_interval,...
245             inter_fig, plot_control.will_pause);
246     end
247
248     iter = iter + 1;
249 end
250
251 % Testing of Trained ANN
252 %*********************************************************************
253 % Identify the best result ANN nodal weights.
254 final_ann_param = fire_pos(1,:,max_iter);
255 % Reshape into ANN weight matrices.
256 Theta1 = reshape(final_ann_param(:,1:theta1_size)',...
257     [hidden_layer_size input_layer_size+1]);
258 Theta2 = reshape(final_ann_param(:,(theta1_size+1):dim_num)',...
259     [num_labels hidden_layer_size+1]);
260 % Evaluate best parameters with the test data set.
261 [results.test_fit,results.test_acc,results.test_pred,...
262     results.test_confuse]=sim_predict_final(Theta1,Theta2,X_test,...
263     y_test, lambda);
```

```
264
265 % Simulation Run Results Archiving
266 %*************************************************************************
267 % Archives any data results not already accounted for. Testing results are
268 % archived in the actual evaluation of the test data; there's no need to
269 % create temporary variables just to archive those results here.
270 results.gbest_fit = fire_fit;
271 results.gbest_pos = fire_pos;
272 results.test_time = toc;
273 results.train_confuse = confuse_mat(:,:,row);
274 results.train_acc = max(ANN_acc(:,max_iter));
275 results.train_pred = ANN_pred(:,size(ANN_pred,2));
276 results.train_fit = best_spark_fit;
277
278 % Plotting of Algorithm Global Fitness Curve
279 %*************************************************************************
280 % Plots the global best fitness if finishing plots are enabled.
281 if(plot_control.final_plot)
282     figure
283     semilogy(fire_fit(:))
284     title('Bare Bones Fireworks Algorithm Fitness Curve')
285     xlabel('Swarm Iterations')
286     ylabel('Fitness Value')
287     fit_log = gcf;
288     save_my_figs(fit_log,filepath,'BBFWA_Fitness_Curve')
289 end
```

B.3.2 Particle Swarm Optimization

```
1  function [results] = Canon_PSO_ANN_sim(X_train,X_test,y_train,y_test,...
2      seed_nnparam,param,pso, plot_control,filepath)
3
4  % Canonical Particle Swarm Optimization Algorithm Artificial Neural ...
5  %   Network Training Algorithm Simulation Variant
6  %
7  % Created By: Patrik Gilley
8  % Date Created: November 27, 2019
9  % Date Last Modified: December 5,2019
10 % Program Name: Canon_PSO_ANN_sim.m
11 %
12 % This program is the final version of the Canonical PSO ANN training
13 % program, reformatted for use in a Monte Carlo simulation framework. This
14 % is the master program for the PSO ANN training algorithm. All of the
15 % neural network programs and functions are used within this program and
16 % its subroutines/functions. The only program on a higher level than this
17 % program is the simulation script itself.
18 %
19 % Program Inputs:
20 %   X_train = Data set sample array that the ANN will be trained with.
21 %   X_test = Data set sample array that the ANN will be tested with.
```

```
22 %   y_train = Data set labels that the ANN will be trained with.
23 %   y_test = Data set labels that the ANN will be tested with.
24 %   seed_nnparam = Pre-trained ANN weights that the ANN that will be
25 %                  trained by this program should be seeded with.
26 %   param = MatLab structure defined in the simulation framework that
27 %           contains the general algorithm settings.
28 %   pso = MatLab structure defined in simulation framework that contains
29 %         PSO algorithm specific settings.
30 %   plot_control = MatLab structure defined in simulation framework that
31 %                  contains confusion matrix plotting settings.
32 %   filepath = String path to directory that figures will be saved to.
33 %
34 % Program Outputs:
35 %   results = MatLab structure that holds the algorithm output data.
36 %   Properties:
37 %       gbest_fit = The array of best global fitness values achieved by the
38 %                   particle swarm.
39 %       gbest_pos = Vector containing the best position achieved by the
40 %                   swarm.
41 %       Training Metrics:
42 %           train_time = Total training time of the CPSO algorithm.
43 %           train_acc = Final training accuracy achieved by the neural
44 %                       network.
45 %           train_fit = Final training fitness achieved by the training
46 %                       algorithm.
47 %           train_pred = Column vector archiving the final training phase
48 %                        output labels.
49 %           train_confuse = Confusion matrix for the gbest ANN predictions
50 %                           on the final evaluation of the training data
51 %                           set.
52 %       Testing Metrics:
53 %           test_time = Total testing time of the CPSO algorithm.
54 %           test_acc = ANN classification accuracy on the testing data set.
55 %           test_fit = CPSO fitness from the testing phase evaluation.
56 %           test_pred = Column vector archiving the output labels from the
57 %                       test phase evaluation.
58 %           test_confuse = Confusion matrix for the gbest ANN predictions
59 %                          on the testing data set.
60 %
61 % User Defined Functions:
62 %   sim_predict_final.m: Function that evaluates the fitness function for
63 %                        the search algorithm.
64 %   confusemat_plot.m: Creates confusion matrix plots to display the
65 %                      progress of the ANN training.
66
67 % Clear Out Previous Figures
68 %*************************************************************************
69 close all
70
71 % Neural Network Parameter Initialization
72 %*************************************************************************
73 % Create training and testing data sets from the parent data set. The MNIST
74 % data set already has pre-defined training and testing data sets; dividing
75 % the data sets further is unnecessary here.
```

```
76
77 % Randomly shuffle the data set.
78 size_train = size(X_train,1);   % Determines number of samples in data set.
79 % Create an integer vector from 1 to size_train with randomly sorted
80 % values.
81 shuffle_train = randperm(size_train,size_train);
82 % Use random integers as new sample indices.
83 X_train(:,:) = X_train(shuffle_train,:);
84 y_train(:,1) = y_train(shuffle_train,:);
85
86 size_test = size(X_test,1);
87 shuffle_test = randperm(size_test,size_test);
88 X_test(:,:) = X_test(shuffle_test,:);
89 y_test(:,:) = y_test(shuffle_test,:);
90
91 % ANN network structure
92 input_layer_size = size(X_train,2);  % Sets input layer size to number of
93                                      % data features in X_train.
94 hidden_layer_size = param.hid_size;
95 num_labels = param.num_lab;          % 10 labels, from 1 to 10.
96                                      % Label 10 represents digit 0.
97
98 % Solution Space and Initial Parameter Initialization
99 %*************************************************************************
100 % Control Structure Parameter Setting
101 upper_bounds = param.uppbound;% Sets the upper bound of the solution space.
102 lower_bounds = param.lowbound;% Sets the lower bound of the solution space.
103 max_iter = param.main_maxiter;% Maximum number of iterations allowed.
104 epsilon_init = param.epsilon_init;% Seed value that sets the range in which
105                                   % the nodal weigths will be initialized.
106 lambda = param.lambda;  % Regularization constant.
107 iter_interval = plot_control.iter_interval;% Sets the interval between
108                                            % intermediate confusion matrix
109                                            % updates.
110 swarm_size = pso.swarm; % Sets the number of PSO particles.
111 c_1 = pso.c1;           % Particle personal best weight.
112 c_2 = pso.c2;           % Particle global best weight.
113 omega = pso.omega;      % Inertial weight, used to set the particle's
114                         % resistance to change in velocities.
115
116 % Internal Parameter Initialization
117 theta1_size = hidden_layer_size*(input_layer_size+1);
118 theta2_size = num_labels*(hidden_layer_size+1);
119 dim_num = theta1_size+theta2_size;   % Uses the ANN structure to determine
120                                      % the number of weights that need to
121                                      % be set. Accounts for bias nodes.
122 iter = 1;   % Initializes an iteration tracking variable.
123 dummy = 0;  % Creates a dummy variable for confusemat_plot.m
124             % figure handle input.
125
126 % Particle Metric Matrix Initialization
127 %*************************************************************************
128 confuse_mat = zeros(num_labels,num_labels,swarm_size);  % Single ...
      iteration of confusion matrices.
```

```
129 ANN_pred = zeros(size(X_train,1),swarm_size); % ANN output labels.
130 ANN_acc = zeros(swarm_size,max_iter);      % ANN classification accuracies.
131
132 % Initializes the particle position matrix to have a row per particle and
133 % a column for every dimension, and a layer for each iteration.
134 part_pos = zeros(swarm_size,dim_num,max_iter);
135 part_vel = zeros(swarm_size,dim_num,max_iter);       % Initializes the ...
       particle velocity matrix with the same organization as the positions.
136 part_best_pos = zeros(swarm_size,dim_num,max_iter);  % Initializes the ...
       best particle position matrix.
137 temp_pos_fit = zeros(swarm_size,1); % Temporary fitness value storage.
138
139 % Initialize the particle best fitness matrix as a column vector.
140 % Matrix is populated with infinity in order to allow for finding of ...
       minimum fitness values without initialization values interfering.
141 part_best_fit = ones(swarm_size,1,max_iter)*Inf;
142
143 % Particle Initialization
144 %**************************************************************************
145 tic;
146 % Randomly determine the sign of the matrix position coordinates.
147 pos_sign = (randi([0,1],swarm_size,dim_num)*2)-1;
148
149 % Seeding code for giving the algorithm a pre-trained ANN by giving an
150 % unrolled vector of nodal weights for the entire network. Clones the
151 % pre-trained ANN weights for each swarm particle.
152 seed_pos = ones(swarm_size,size(seed_nnparam,1)).*seed_nnparam';
153
154 % Seeds the input to hidden layer nodal weights in particles. Adds random
155 % numbers to scatter particles.
156 part_pos(:,1:theta1_size,1) = ...
       seed_pos(:,1:theta1_size)+pos_sign(:,1:theta1_size)...
157     .*(epsilon_init*2*rand(swarm_size,theta1_size)-epsilon_init);
158
159 % Seeds the hidden to output layer nodal weights in particles.
160 part_pos(:,(theta1_size+1):dim_num,1)=seed_pos(:,...
161     (theta1_size+1):dim_num)+pos_sign(:,(theta1_size+1):dim_num).*...
162     (epsilon_init*2*rand(swarm_size,(dim_num-theta1_size))-epsilon_init);
163
164 % Archive the particle starting positions as their personal best.
165 part_best_pos(:,:,1) = part_pos(:,:,1);
166
167 % Determine each particle personal best solution value/fitness.
168 % Acquires the data for the Theta1 matrix parameters, and reshape to
169 % properly fit ANN structure.
170 Theta1 = reshape(part_best_pos(:,1:theta1_size)',...
171     [hidden_layer_size input_layer_size+1 swarm_size]);
172
173 % The position data is transposed to take advantage of the reshape
174 % function's behavior, which preserves the columnwise ordering of data.
175 Theta2 = reshape(part_best_pos(:,(theta1_size+1):dim_num)',...
176     [num_labels hidden_layer_size+1 swarm_size]);
177
178 % Runs each particle position through the fitness function and stores the
```

```
% results for comparison.
for ann_fit = 1:swarm_size
    [part_best_fit(ann_fit,:,1),ANN_acc(ann_fit,1),ANN_pred(:,ann_fit),...
        confuse_mat(:,:,ann_fit)]=sim_predict_final(Theta1(:,:,ann_fit),...
        Theta2(:,:,ann_fit),X_train,y_train,lambda);
end

% Determine the global best fitness position and value.
%*********************************************************************
% Finds the smallest particle fitness value and its index.
[best_fit, best_ind] = min(part_best_fit(:));

% Converts the index reported by the min() command into a useable index.
[row, ¬] = ind2sub(size(part_best_fit),best_ind);

% Pulls the position of the best fit particle.
global_best_pos(1,:,1) = part_pos(row,:,1);

% Hold the best fitness value for later comparison.
global_best_fit(1,:,1) = best_fit;

% If plots are enabled, a confusion matrix will be plotted for the global
% best particle solution.
if(plot_control.will_plot)
    sol_select = row;
    ann_fit = sol_select;
    [inter_fig] = confusemat_plot(confuse_mat(:,:,ann_fit), sol_select,...
        iter,ann_fit,max_iter,iter_interval,dummy,plot_control.will_pause);
end

% PSO Algorithm Execution
%*********************************************************************
while(iter < max_iter)

    % Particle Velocity and Position Update
    %*****************************************************************
    % Finds the particle's next velocity and position using its current
    % position and velocity.
    part_vel(:,:,iter+1) = omega*(part_vel(:,:,iter)) + c_1*...
        rand(swarm_size,dim_num).*(part_best_pos(:,:,iter)-...
        (part_pos(:,:,iter))) + c_2*rand(swarm_size,dim_num).*...
        (global_best_pos(1,:,iter)-(part_pos(:,:,iter)));

    % Update each dimension location of a particle.
    part_pos(:,:,iter+1) = part_pos(:,:,iter) + part_vel(:,:,iter+1);

    % Out of Function Bounds Detection and Correction
    %*****************************************************************
    % Finds any particle positions that have gone over the upper limit.
    up_check = part_pos(:,:,iter+1) > upper_bounds;

    % Finds any particle positions that have exceeded the lower limit.
    down_check = part_pos(:,:,iter+1) < lower_bounds;

```

```
233     % Split into two if statements to ensure that both positive and
234     % negative boundaries are checked.
235
236     % Check to see if the particle has exceeded the max positive bound.
237     if(ismember(1,up_check))
238         % Set the new velocity to be the opposite of the old velocity.
239         part_vel(:,:,iter+1) = (¬up_check.*part_vel(:,:,iter+1))+...
240             (up_check.*(-part_vel(:,:,iter+1)));
241         % Force the particle position to the max positive position.
242         part_pos(:,:,iter+1) = (¬up_check.*part_pos(:,:,iter+1))+...
243             (up_check.*upper_bounds);
244     end
245
246     % Checks to see if the particle has exceeded the max negative bound.
247     if(ismember(1,down_check))
248         % Set the new velocity to be the opposite of the old velocity.
249         part_vel(:,:,iter+1) = (¬down_check.*part_vel(:,:,iter+1))+...
250             (down_check.*(-part_vel(:,:,iter+1)));
251         % Force the particle position to the max negative position.
252         part_pos(:,:,iter+1) = (¬down_check.*part_pos(:,:,iter+1))+...
253             (down_check.*lower_bounds);
254     end
255
256     % Particle Fitness Evaluation and Update
257     %*********************************************************************
258     % Check particle position vs. best position so far.
259
260     % Acquire the data for the Theta1 matrix parameters, and reshape to
261     % properly fit ANN structure.
262     Theta1 = reshape(part_pos(:,1:theta1_size,iter+1)',...
263         [hidden_layer_size input_layer_size+1 swarm_size]);
264
265     Theta2 = reshape(part_pos(:,(theta1_size+1):dim_num,iter+1)',...
266         [num_labels hidden_layer_size+1 swarm_size]);
267
268     % Runs each particle position through the function that is being
269     % minimized and stores the results for comparison.
270     for ann_fit = 1:swarm_size
271         [temp_pos_fit(ann_fit,:,1),ANN_acc(ann_fit,iter+1),...
272             ANN_pred(:,ann_fit),confuse_mat(:,:,ann_fit)] =...
273             sim_predict_final(Theta1(:,:,ann_fit),Theta2(:,:,ann_fit),...
274             X_train,y_train,lambda);
275     end
276
277     % Find all of the fitness values that have decreased.
278     new = temp_pos_fit < part_best_fit(:,:,iter);
279
280     % Check to see if the new fitness values have decreased at all.
281     if(ismember(1,new))
282         % Find the fitness values that have decreased and add in the values
283         % that are better than the current ones.
284         part_best_pos(:,:,iter+1) = (new.*part_pos(:,:,iter+1))+...
285             (part_best_pos(:,:,iter).*¬new);
286         % Assign the new fitness value in the next iteration layer.
```

```
287         part_best_fit(:,:,iter+1) = (new.*temp_pos_fit(:,:))+...
288             (part_best_fit(:,:,iter).*¬new);
289     else
290         % Keeps the current best position and fitness values if the new
291         % solution is worse than the previous one.
292         part_best_pos(:,:,iter+1) = part_best_pos(:,:,iter);
293         part_best_fit(:,:,iter+1) = part_best_fit(:,:,iter);
294     end
295
296     % Global Function Fitness Check and Update
297     %*********************************************************************
298     % Finds the smallest fitness value of the particles and its index.
299     [hold_fit, hold_ind] = min(part_best_fit(:,:,iter+1));
300
301     % Check the current position against the global best position. This
302     % uses the same general approach of the particle personal best
303     % fitnesses.
304     if(hold_fit < global_best_fit(1,1,iter))
305         global_best_pos(1,:,iter+1) = part_pos(hold_ind,:,iter+1);
306         global_best_fit(1,1,iter+1) = hold_fit;
307     else
308         global_best_pos(1,:,iter+1) = global_best_pos(1,:,iter);
309         global_best_fit(1,1,iter+1) = global_best_fit(1,1,iter);
310     end
311
312     % If plots are enabled, a confusion matrix will be plotted for the
313     % current global best particle solution.
314     if(plot_control.will_plot)
315         sol_select = hold_ind;
316         ann_fit = sol_select;
317         [inter_fig] = confusemat_plot(confuse_mat(:,:,ann_fit),...
318             sol_select, iter, ann_fit, max_iter, iter_interval,...
319             inter_fig, plot_control.will_pause);
320     end
321
322     iter = iter + 1;    % Advances the iteration counter.
323 end
324
325 results.train_time = toc; % Archives the training time.
326
327 % Testing of Trained ANN
328 %*************************************************************************
329 % Identify the best result ANN nodal weights.
330 final_ann_param = global_best_pos(1,:,max_iter);
331
332 % Reshape final best ANN weights into weight matrices.
333 Theta1 = reshape(final_ann_param(:,1:theta1_size)',...
334     [hidden_layer_size input_layer_size+1]);
335
336 Theta2 = reshape(final_ann_param(:,(theta1_size+1):dim_num)',...
337     [num_labels hidden_layer_size+1]);
338
339 % Evaluate best ANN weights using test data set.
340 [results.test_fit,results.test_acc,results.test_pred,...
```

```
341     results.test_confuse]=sim_predict_final(Theta1,Theta2,X_test,...
342     y_test,lambda);
343 
344 % Simulation Run Results Archiving
345 %*************************************************************************
346 % Archives any data results not already accounted for. Testing results are
347 % archived in the actual evaluation of the test data; there's no need to
348 % create temporary variables just to archive those results here.
349 results.gbest_fit = global_best_fit;
350 results.gbest_pos = global_best_pos;
351 results.test_time = toc;
352 results.train_confuse = confuse_mat(:,:,hold_ind);
353 results.train_acc = max(ANN_acc(:,max_iter));
354 results.train_pred = ANN_pred(:,size(ANN_pred,2));
355 results.train_fit = hold_fit;
356 
357 % Plotting of Algorithm Global Fitness Curve
358 %*************************************************************************
359 % Plots the global best fitness if finishing plots are enabled.
360 if(plot_control.final_plot)
361     figure
362     semilogy(global_best_fit(:))
363     title(['Canonical Particle Swarm Optimization Global Fitness'...
364         'Logarithmic Plot'])
365     xlabel('Swarm Iterations')
366     ylabel('Fitness Value')
367     fit_log = gcf;
368     save_my_figs(fit_log,filepath,'CPSO_Fitness_Curve')
369 end
```

B.3.3 Cooperative Particle Swarm Optimization

```
1  function [results] = Coop_CPSO_ANN_esplit_sim(X_train,X_test,y_train,...
2      y_test,seed_nnparam,param,pso,plot_control,filepath)
3  
4  % Cooperative Canonical Particle Swarm Optimization Artificial Neural
5  % Network Training Algorithm Simulation Variant
6  %
7  % Created By: Patrik Gilley
8  % Date Created: November 29, 2019
9  % Date Last Modified: December 13,2019
10 % Program Name: Coop_CPSO_ANN_esplit_sim.m
11 %
12 % Program Inputs:
13 %   X_train = Data set sample array that the ANN will be trained with.
14 %   X_test = Data set sample array that the ANN will be tested with.
15 %   y_train = Data set labels that the ANN will be trained with.
16 %   y_test = Data set labels that the ANN will be tested with.
17 %   param = MatLab structure defined in the simulation framework that
18 %           contains the general algorithm settings.
```

```
19 %   pso = MatLab structure defined in simulation framework that contains
20 %         CPSO algorithm specific settings.
21 %   plot_control = MatLab structure defined in simulation framework that
22 %                  contains confusion matrix plotting settings.
23 %   filepath = String path to directory that figures will be saved to.
24 %
25 % Program Outputs:
26 %   results = MatLab structure that holds the algorithm output data.
27 %   Properties:
28 %       gbest_fit = The array of best global fitness values achieved by the
29 %                   particle swarm.
30 %       gbest_pos = Vector containing the best position achieved by the
31 %                   swarm.
32 %       train_acc = Final training accuracy achieved by the neural network.
33 %       train_fit = Final training fitness achieved by the training
34 %                   algorithm.
35 %       train_pred = Column vector archiving the final training phase
36 %                    output labels.
37 %       train_time = Total training time of the Coop Esplit PSO algorithm.
38 %       test_acc = ANN classification accuracy on the testing data set.
39 %       test_fit = CPSO fitness from the testing phase evaluation.
40 %       test_pred = Column vector archiving the output labels from the test
41 %                   phase evaluation.
42 %       test_time = Total testing time of the Coop Esplit PSO algorithm.
43 %
44 % This program is the final version of the Cooperative PSO ANN training
45 % program, reformatted for use in a Monte Carlo simulation framework. This
46 % is the Cooperative PSO algorithm variant that utilizes the Esplit
47 % architecture from the source paper.
48 %
49 % User Defined Functions:
50 %   sim_predict_final.m: Function that evaluates the fitness function for
51 %                        the search algorithm.
52 %   confusemat_plot.m: Creates confusion matrix plots to display the
53 %                      progress of the ANN training.
54
55 % Clear Out Previous Figures
56 %*************************************************************************
57 close all
58
59 % Neural Network Parameter Initialization
60 %*************************************************************************
61 % Randomly shuffle the data set.
62 size_train = size(X_train,1);
63 shuffle_train = randperm(size_train,size_train);
64 X_train(:,:) = X_train(shuffle_train,:);
65 y_train(:,1) = y_train(shuffle_train,:);
66
67 size_test = size(X_test,1);
68 shuffle_test = randperm(size_test,size_test);
69 X_test(:,:) = X_test(shuffle_test,:);
70 y_test(:,:) = y_test(shuffle_test,:);
71
72 % ANN network structure
```

```
input_layer_size = size(X_train,2);  % Sets input layer size to number of
                                     % data features in X_train.
hidden_layer_size = param.hid_size;
num_labels = param.num_lab;          % 10 labels, from 1 to 10.
                                     % Label 10 represents 0.

% Solution Space and Initial Parameter Initialization
%*************************************************************************
% Control Structure Parameter Setting
upper_bounds = param.uppbound;% Sets the upper bound of the solution space.
lower_bounds = param.lowbound;% Sets the lower bound of the solution space.
max_iter = param.coop_maxiter;% Maximum number of iterations allowed.
epsilon_init = param.epsilon_init;% Seed value that sets the range in which
                                  % the nodal weigths will be initialized.
lambda = param.lambda;        % Regularization constant.
iter_interval = plot_control.iter_interval;% Sets the interval between
                                           % intermediate confusion matrix
                                           % updates.
% NOTE: Swarm hyperparameters are applied equally across all sub-swarms;
% currently no method of setting parameters for each individual sub-swarm.
swarm_size = pso.swarm; % Sets the number of CPSO sub-swarm particles.
omega = pso.omega;      % Inertial weight, used to set the particle's
                        % resistance to change in velocities.
c_1 = pso.c1;           % Particle personal best weight.
c_2 = pso.c2;           % Particle global best weight.

% Internal Parameter Initialization
theta1_size = hidden_layer_size*(input_layer_size+1);
theta2_size = num_labels*(hidden_layer_size+1);
dim_num = theta1_size+theta2_size;  % Uses the ANN structure to determine
                                    % the number of weights that need
                                    % to be set. Accounts for bias nodes.
split_dim = dim_num/2;  % Determines the number of dimensions an even
                        % split of the context vector means.
odd_test = mod(dim_num,2);  % Checks to see if the resulting value is even.

% If overall number of dimensions is odd, then the context vector
% cannot be evenly split. The extra dimension is given to sub-swarm 2.
if(odd_test)
    split_dim1 = floor(split_dim);
    split_dim2 = ceil(split_dim);
else
    split_dim1 = dim_num/2;
    split_dim2 = split_dim1;
end
iter = 1;     % Initializes an iteration tracking variable.
dummy = 0;    % Creates a dummy variable for confusemat_plot.m figure
              % handle input.

% Particle Metric Matrix Initialization
%*************************************************************************
% Subswarm 1 matrix structure.
context = zeros(1,dim_num);    % Initializes the context vector of the
                               % CPSO sub-swarms.
```

```
% Initialize the particle position matrix to have a row per particle and
% a column for every dimension, and a layer for each iteration.
coop.swarm1.part_pos = zeros(swarm_size,split_dim1,max_iter);
% Initialize the particle velocity matrix with the same organization as
% the positions.
coop.swarm1.part_vel = zeros(swarm_size,split_dim1,max_iter);
% Initialize the best particle position matrix.
coop.swarm1.part_best_pos = zeros(swarm_size,split_dim1,max_iter);
% Initialize the particle best fitness matrix as a column vector.
% Matrix is populated with infinity in order to allow for finding of
% minimum fitness values without initialization values interfering.
coop.swarm1.part_best_fit = ones(swarm_size,1,max_iter)*Inf;
% Initialize global best position and fitness matrices.
coop.swarm1.global_best_pos = zeros(1,split_dim1,max_iter);
coop.swarm1.global_best_fit = zeros(1,1,max_iter);
% Store the number of dimensions that sub-swarm 1 is responsible for.
coop.swarm1.theta_size = split_dim1;

% Subswarm 2 matrix structure. Uses same scheme aas sub-swarm 1.
coop.swarm2.part_pos = zeros(swarm_size,split_dim2,max_iter);
coop.swarm2.part_vel = zeros(swarm_size,split_dim2,max_iter);
coop.swarm2.part_best_pos = zeros(swarm_size,split_dim2,max_iter);
coop.swarm2.part_best_fit = ones(swarm_size,1,max_iter)*Inf;
coop.swarm2.global_best_pos = zeros(1,split_dim2,max_iter);
coop.swarm2.global_best_fit = zeros(1,1,max_iter);
coop.swarm2.theta_size = split_dim2;

% General matrices
% Context vector position and fitness matrices.
global_best_fit = zeros(1,1,max_iter);
global_best_pos = zeros(1,dim_num,max_iter);

% Matrix to hold confusion matrices generated in fitness
% function evaluation.
confuse_mat = zeros(num_labels,num_labels,swarm_size);

% Matrix to hold the ANN output label predictions.
ANN_pred = zeros(size(X_train,1),swarm_size);

% Matrix to hold ANN classification accuracy measurements calculated during
% fitness function evaluation.
ANN_acc = zeros(swarm_size,max_iter);
temp_pos_fit = zeros(swarm_size,1);

% Sub-Swarm Initialization
%**************************************************************************
tic;
% Clones the pre-trained ANN weights for each swarm particle.
seed_pos = ones(swarm_size,size(seed_nnparam,1)).*seed_nnparam';

% Sub-swarm 1 (Theta 1 Matrix)
pos_sign = (randi([0,1],swarm_size,split_dim1)*2)-1;

% Initialize the first subswarm weights to the Theta 1 matrix dimensions.
```

```
coop.swarm1.part_pos(:,:,1) = seed_pos(:,1:split_dim1)+pos_sign.*...
    (epsilon_init*2*rand(swarm_size,split_dim1)-epsilon_init);
% Saves the current subswarm positions as best positions.
coop.swarm1.part_best_pos(:,:,1) = coop.swarm1.part_pos(:,:,1);

% Sub-swarm 2 (Theta 2 Matrix)
pos_sign = (randi([0,1],swarm_size,split_dim2)*2)-1;

% Initialize the second subswarm weights to the Theta 2 matrix dimensions.
coop.swarm2.part_pos(:,:,1) = seed_pos(:,(split_dim1+1):dim_num)+...
    pos_sign.*(epsilon_init*2*rand(swarm_size,split_dim2)-epsilon_init);
% Save the current subswarm positions as best positions.
coop.swarm2.part_best_pos(:,:,1) = coop.swarm2.part_pos(:,:,1);

% Corrects for the odd split's extra dimension so that the current context
% vector indexing scheme can still work with it.
if(odd_test)
   split_dim2 = split_dim2-1;
end

% Particle Personal Best Fitness Determination
%**************************************************************************
% For initialization, the initial fitness will be determined by pairing up
% particle from each subswarm using their indices; e.g. particle 1 from
% sub-swarm 1 will be evaluated with particle 1 from sub-swarm 2.
for ann_fit = 1:swarm_size
    % Context Vector Construction
    %**********************************************************************
    % Pull the j-ith particle from sub-swarm 1 for evaluation.
    context(1,1:split_dim1) = coop.swarm1.part_best_pos(ann_fit,:,1);
    % Pulls the j-ith particle from sub-swarm 2 (matching sub1's index)
    % for evaluation.
    context(1,(split_dim2+1):dim_num) = ...
        coop.swarm2.part_best_pos(ann_fit,:,1);

    % Acquires the data for the Theta1 matrix parameters, and reshapes
    % both sub-swarm particles to properly fit ANN structure.
    Theta1 = reshape(context(1,1:theta1_size)',...
        [hidden_layer_size input_layer_size+1]);
    Theta2 = reshape(context(1,(theta1_size+1):dim_num)',...
        [num_labels hidden_layer_size+1]);

    % Particle Pair Fitness Evaluation
    %**********************************************************************
    % Acquire current context vector fitness.
    [initial_tempfit,ANN_acc(ann_fit,1),ANN_pred(:,ann_fit),...
        confuse_mat(:,:,ann_fit)] = sim_predict_final(Theta1,Theta2,...
        X_train,y_train,lambda);
    % Assign evaluated fitness to all sub-swarm particles used in the
    % current context vector.
    coop.swarm1.part_best_fit(ann_fit,:,1) = initial_tempfit;
    coop.swarm2.part_best_fit(ann_fit,:,1) = initial_tempfit;
end

```

```
% Determine the global best fitness position and value.
%*************************************************************************
for sub_swarm = 1:2
    % Find the smallest fitness value and index of the particles.
    [best_fit, best_ind] = min(coop.(['swarm' num2str(sub_swarm)])...
        .part_best_fit(:));
    % Convert best_ind into row and column indices.
    [row, ~] = ind2sub(size(coop.(['swarm' num2str(sub_swarm)])...
        .part_best_fit),best_ind);

    % Pull the position and fitness of the gbest particle for
    % each sub-swarm.
    coop.(['swarm' num2str(sub_swarm)]).global_best_pos(1,:,1) = ...
        coop.(['swarm' num2str(sub_swarm)]).part_pos(row,:,1);
    coop.(['swarm' num2str(sub_swarm)]).global_best_fit(1,:,1) = best_fit;

    % Assigns gbest sub-swarm weights to context vector for record-keeping.
    if(sub_swarm == 1)
        context(1,1:split_dim1) = coop.swarm1.global_best_pos(1,:,1);
    else
        context(1,(split_dim2+1):dim_num) =...
            coop.swarm2.global_best_pos(1,:,1);
    end
end

% Global best fitness and weight archiving. Gbest fitness values for the
% context vector should be the same for both sub-swarms.
global_best_fit(1,:,1) = coop.swarm1.global_best_fit(1,:,1);
global_best_pos(1,:,1) = [coop.swarm1.global_best_pos(1,:,1)...
    coop.swarm2.global_best_pos(1,:,1)];

% Runs each particle position through the fitness function and stores the
% results for comparison.
if(plot_control.will_plot)
    sol_select = row;
    ann_fit = sol_select;
    [inter_fig] = confusemat_plot(confuse_mat(:,:,ann_fit), sol_select,...
        iter, ann_fit, max_iter, iter_interval, dummy,...
        plot_control.will_pause);
end

% PSO Algorithm Execution
%*************************************************************************
while(iter < max_iter)
    for sub_swarm = 1:2
        % Particle Velocity and Position Update
        %*****************************************************************
        % Finds the particle's next velocity and position using its current
        % position and velocity.
        coop.(['swarm' num2str(sub_swarm)]).part_vel(:,:,iter+1)=omega*...
            (coop.(['swarm' num2str(sub_swarm)]).part_vel(:,:,iter))...
            +c_1*rand(swarm_size,...
            coop.(['swarm' num2str(sub_swarm)]).theta_size).*...
            (coop.(['swarm' num2str(sub_swarm)]).part_best_pos(:,:,iter)...
```

```
289             -(coop.(['swarm' num2str(sub_swarm)]).part_pos(:,:,iter)))...
290             +c_2*rand(swarm_size,coop.(['swarm' num2str(sub_swarm)]))...
291             .theta_size).*(coop.(['swarm' num2str(sub_swarm)])...
292             .global_best_pos(1,:,iter)-(coop.(['swarm'...
293             num2str(sub_swarm)]).part_pos(:,:,iter)));
294
295         % Updates each dimension location of a particle.
296         coop.(['swarm' num2str(sub_swarm)]).part_pos(:,:,iter+1)=...
297             coop.(['swarm' num2str(sub_swarm)]).part_pos(:,:,iter)+...
298             coop.(['swarm' num2str(sub_swarm)]).part_vel(:,:,iter+1);
299
300         % Out of Function Bounds Detection and Correction
301         %*****************************************************************
302         % Find any particle positions that have gone over the upper limit.
303         up_check = coop.(['swarm' num2str(sub_swarm)])...
304             .part_pos(:,:,iter+1) > upper_bounds;
305         % Find any particle positions that have exceeded the lower limit.
306         down_check = coop.(['swarm' num2str(sub_swarm)])...
307             .part_pos(:,:,iter+1) < lower_bounds;
308
309         % Check to see if the particle has exceeded the max positive bound.
310         if(ismember(1,up_check))
311             % Set the new velocity to be the opposite of the old velocity.
312             coop.(['swarm' num2str(sub_swarm)]).part_vel(:,:,iter+1) = ...
313                 (¬up_check.*coop.(['swarm' num2str(sub_swarm)])...
314                 .part_vel(:,:,iter+1))+(up_check.*(-coop.(['swarm'...
315                 num2str(sub_swarm)]).part_vel(:,:,iter+1)));
316             % Force the particle position to the max positive position.
317             coop.(['swarm' num2str(sub_swarm)]).part_pos(:,:,iter+1) =...
318                 (¬up_check.*coop.(['swarm' num2str(sub_swarm)])...
319                 .part_pos(:,:,iter+1))+(up_check.*upper_bounds);
320         end
321
322         % Check to see if the particle has exceeded the max negative bound.
323         if(ismember(1,down_check))
324             % Set the new velocity to be the opposite of the old velocity.
325             coop.(['swarm' num2str(sub_swarm)]).part_vel(:,:,iter+1) =...
326                 (¬down_check.*coop.(['swarm' num2str(sub_swarm)])...
327                 .part_vel(:,:,iter+1))+(down_check.*(-coop.(['swarm'...
328                 num2str(sub_swarm)]).part_vel(:,:,iter+1)));
329             % Forces the particle position to the max negative position.
330             coop.(['swarm' num2str(sub_swarm)]).part_pos(:,:,iter+1) =...
331                 (¬down_check.*coop.(['swarm' num2str(sub_swarm)])...
332                 .part_pos(:,:,iter+1))+(down_check.*lower_bounds);
333         end
334
335         % Particle Fitness Evaluation and Update
336         %*****************************************************************
337         % Determine the fitness values of each sub-swarm particle.
338         for ann_fit = 1:swarm_size
339             % Sub Swarm Dependent Context Vector Construction
340             %*************************************************************
341             % If working through sub-swarm 1, use best sub2 weights and
342             % cycle through all sub1 weights.
```

```
343             if(sub_swarm == 1)
344                 context(1,1:split_dim1) =...
345                     coop.swarm1.part_pos(ann_fit,:,iter+1);
346                 context(1,(split_dim2+1):dim_num) =...
347                     coop.swarm2.global_best_pos(1,:,iter);
348             % If working through sub-swarm 2, use best sub1 weights and
349             % cycle through all sub2 weights.
350             elseif(sub_swarm == 2)
351                 context(1,1:split_dim1) =...
352                     coop.swarm1.global_best_pos(1,:,iter);
353                 context(1,(split_dim2+1):dim_num) =...
354                     coop.swarm2.part_pos(ann_fit,:,iter+1);
355             end
356
357             % Context Vector Evaluation
358             %*************************************************************
359             % Form ANN weight matrices from the sub-swarm particles.
360             Theta1 = reshape(context(:,1:theta1_size)',...
361                 [hidden_layer_size input_layer_size+1]);
362             Theta2 = reshape(context(:,(theta1_size+1):dim_num)',...
363                 [num_labels hidden_layer_size+1]);
364
365             % Run each particle position through the function that is
366             % being minimized and store the results for comparison.
367             [temp_pos_fit(ann_fit,:,1)]=sim_predict_final(Theta1,Theta2,...
368                 X_train,y_train,lambda);
369         end
370
371         % Sub-Swarm Particle Best Fitness Check and Update
372         %*****************************************************************
373         % Check to see if any sub-swarm particle positions found are
374         % better than the previous generation.
375         new = temp_pos_fit < coop.(['swarm' num2str(sub_swarm)])...
376             .part_best_fit(:,:,iter);
377
378         % Check to see if the any fitness values have decreased.
379         if(ismember(1,new))
380            % Find the fitness values that have decreased and add any old
381            % values that are better than the current ones.
382            coop.(['swarm' num2str(sub_swarm)]).part_best_pos(:,:,iter+1)...
383                =(new.*coop.(['swarm' num2str(sub_swarm)])...
384                .part_pos(:,:,iter+1))+(coop.(['swarm'...
385                num2str(sub_swarm)]).part_best_pos(:,:,iter).*¬new);
386
387            % Assign the new fitness value in the next iteration layer.
388            coop.(['swarm' num2str(sub_swarm)]).part_best_fit(:,:,iter+1)...
389                =(new.*temp_pos_fit(:,:))+(coop.(['swarm'...
390                num2str(sub_swarm)]).part_best_fit(:,:,iter).*¬new);
391         else
392            % Keep the current best position and fitness values if all of
393            % the new solutions are worse than the previous ones.
394            coop.(['swarm' num2str(sub_swarm)]).part_best_pos(:,:,iter+1)...
395               =coop.(['swarm' num2str(sub_swarm)]).part_best_pos(:,:,iter);
396
```

```
397             coop.(['swarm' num2str(sub_swarm)]).part_best_fit(:,:,iter+1)...
398                 =coop.(['swarm' num2str(sub_swarm)]).part_best_fit(:,:,iter);
399         end
400
401         % Sub-Swarm Global Function Fitness Check and Update
402         %*****************************************************************
403         % Find the smallest fitness value and index of the particles.
404         [hold_fit, hold_ind] = min(coop.(['swarm' num2str(sub_swarm)])...
405             .part_best_fit(:,:,iter+1));
406
407         % If the sub-swarm 1's best pbest solution is better than the gbest
408         % solution, update gbest.
409         if(hold_fit < coop.(['swarm' num2str(sub_swarm)])...
410                 .global_best_fit(1,1,iter) && sub_swarm == 1)
411             % Update gbest fitness and position.
412             coop.swarm1.global_best_pos(1,:,iter+1) = coop.swarm1...
413                 .part_pos(hold_ind,:,iter+1);
414             coop.swarm1.global_best_fit(1,1,iter+1) = hold_fit;
415
416             % Sub swarm credit assignment strategy; if a new global
417             % best is found, then update the fitness values of all best
418             % particles that generated the new global best.
419
420             % Update the global best fitness value of the second subswarm.
421             coop.swarm2.global_best_fit(1,1,iter+1) = hold_fit;
422             % Updates the corresponding best particle fitness that matches
423             % the global best (Updates the list item that global best was
424             % selected from).
425             coop.swarm2.part_best_fit(hold_ind,1,iter+1) = hold_fit;
426
427         % If the sub-swarm 2's best pbest solution is better than the gbest
428         % solution, update gbest.
429         elseif(hold_fit < coop.(['swarm' num2str(sub_swarm)])...
430                 .global_best_fit(1,1,iter) && sub_swarm == 2)
431             % Update gbest fitness and position.
432             coop.swarm2.global_best_pos(1,:,iter+1) = coop.swarm2...
433                 .part_pos(hold_ind,:,iter+1);
434             coop.swarm2.global_best_fit(1,1,iter+1) = hold_fit;
435
436             % Sub swarm credit assignment strategy; if a new global
437             % best is found, then update the fitness values of all best
438             % particles that generated the new global best.
439
440             % Update the global best fitness value of the first subswarm.
441             coop.swarm1.global_best_fit(1,1,iter+1) = hold_fit;
442             % Updates the corresponding best particle fitness that matches
443             % the global best (Updates the list item that global best was
444             % selected from).
445             coop.swarm1.part_best_fit(hold_ind,1,iter+1) = hold_fit;
446         else
447             % If no better solution found, preserve current gbest position.
448             coop.(['swarm' num2str(sub_swarm)])...
449                 .global_best_pos(1,:,iter+1) = coop...
450                 .(['swarm' num2str(sub_swarm)]).global_best_pos(1,:,iter);
```

```
451             % If no better solution found, preserve gbest fitness.
452             coop.(['swarm' num2str(sub_swarm)])...
453                 .global_best_fit(1,1,iter+1) = coop...
454                 .(['swarm' num2str(sub_swarm)]).global_best_fit(1,1,iter);
455         end
456     end
457
458     % Context Vector (Overall Global Best) Update
459     %*****************************************************************
460     % This code ensures that the values in the context vector represent
461     % the global best particle positions from the current iteration
462     % being considered, and then evalautes the context vector to
463     % provide an overall picture of improvement.
464
465     % Build context vector from the sub-swarm gbest particles.
466     context(1,1:split_dim1) = coop.swarm1.global_best_pos(1,:,iter+1);
467     context(1,(split_dim2+1):dim_num) = ...
468         coop.swarm2.global_best_pos(1,:,iter+1);
469
470     % Reshape the context vector to properly fit ANN structure.
471     Theta1 = reshape(context(:,1:theta1_size)',...
472         [hidden_layer_size input_layer_size+1]);
473     Theta2 = reshape(context(:,(theta1_size+1):dim_num)',...
474         [num_labels hidden_layer_size+1]);
475
476     % Evaulate context vector fitness.
477     [global_best_fit(1,1,iter+1),ANN_acc(ann_fit,iter+1),...
478         ANN_pred(:,ann_fit),confuse_mat(:,:,ann_fit)]=sim_predict_final(...
479         Theta1,Theta2,X_train,y_train,lambda);
480     % Archive the current gbest context vector.
481     global_best_pos(1,:,iter+1) = context(1,:);
482
483     % If plots are enabled, a confusion matrix will be plotted for the
484     % current global best particle solution.
485     if(plot_control.will_plot)
486         sol_select = hold_ind;
487         ann_fit = sol_select;
488         [inter_fig]=confusemat_plot(confuse_mat(:,:,ann_fit),sol_select,...
489             iter,ann_fit,max_iter,iter_interval,inter_fig,...
490             plot_control.will_pause);
491     end
492
493     iter = iter + 1;    % Advances the iteration counter.
494 end
495
496 results.train_time = toc;
497
498 % Testing of Trained ANN
499 %*********************************************************************
500 % Identify the best result ANN nodal weights.
501 final_ann_param = global_best_pos(1,:,max_iter);
502 % Reshape best parameter weights into ANN weight matrices.
503 Theta1 = reshape(final_ann_param(:,1:theta1_size)',...
504     [hidden_layer_size input_layer_size+1]);
```

```
505 Theta2 = reshape(final_ann_param(:,(theta1_size+1):dim_num)',...
506     [num_labels hidden_layer_size+1]);
507 % Evaluate best ANN configuration using the test data set.
508 [results.test_fit,results.test_acc,results.test_pred,¬] =...
509     sim_predict_final(Theta1, Theta2, X_test, y_test, lambda);
510
511 % Simulation Run Results Archiving
512 %*************************************************************************
513 % Archives any data results not already accounted for. Testing results are
514 % archived in the actual evaluation of the test data; there's no need to
515 % create temporary variables just to archive those results here.
516 results.gbest_fit = global_best_fit;
517 results.gbest_pos = global_best_pos;
518 results.test_time = toc;
519 results.train_acc = max(ANN_acc(:,max_iter));
520 results.train_pred = ANN_pred(:,size(ANN_pred,2));
521 results.train_fit = hold_fit;
522 results.swarm1_fit = coop.swarm1.part_best_fit;
523 results.swarm2_fit = coop.swarm2.part_best_fit;
524
525 % Plotting of Algorithm Global Fitness Curve
526 %*************************************************************************
527 % Plots the global best fitness if finishing plots are enabled.
528 if(plot_control.final_plot)
529     % Plots the global best fitness
530     figure
531     semilogy(global_best_fit(:))
532     title(['Esplit Cooperative Particle Swarm Optimization Global'...
533         'Fitness Logarithmic Plot'])
534     xlabel('Swarm Iterations')
535     ylabel('Fitness Value')
536     fit_log = gcf;
537     save_my_figs(fit_log,filepath,'Esplit_Coop_PSO_Fitness_Curve')
538 end
```

B.4 Sub-Function Code

B.4.1 Fitness Functions

```
1  function [fit,ANN_acc,p,confuse_mat]=sim_predict_final(Theta1,Theta2,X,...
2      y,lambda)
3
4  % Search Algorithm Fitness Calculator for OCR MLP ANN
5  % Created by: Patrik Gilley
6  % Date Created: November 27, 2019
7  % Date Last Modified: December 5, 2019
8  % Program Name: sim_predict_final.m
9  %
10 % This program was created to take the trained weights of an ANN and use
```

```
% them to predict the label of the data inputs to that ANN. Given a
% database of handwritten digits and the nodal weights of an ANN trained
% for classifying those digits, this program would predict the output label
% of each input image. Currently, this program is set for a specified ANN
% layer structure of one input layer, one hidden layer, and one output
% layer. This program is set for a classification ANN that uses the sigmoid
% function for its activation function. This is a copy of sim_predict.m,
% created to clean up older code while preserving what code is useful.
%
% Program Inputs:
%   Theta1 = The nodal weight matrix for the input to hidden layer
%            connections.
%   Theta2 = The nodal weight matrix for the hidden to output layer
%            connections.
%   X = The data inputs to the neural network.
%   y = The labels that correspond to the data inputs used with the ANN.
%   lambda = The lambda or regularization constant that the weight
%            regularization calculation should use.
%
% Program Outputs:
%   fit = Overall fitness value of the ANN nodal weights, based on training
%         accuracy.
%   ANN_acc = The accuracy of the ANN on the provided data with the current
%             nodal weights.
%   p = Vector of labels that match the highest output probability of the
%       ANN.
%   confuse_mat = Array of confusion matrix values generated from the
%                 fitness value evaluation.
%
% NOTE: The code partially adapts code from a Coursera online course about
%       machine learning created by Dr. Andrew Ng.

% Feed Training Data Through ANN
%*************************************************************************
% Identify the number of input data entries to correct for bias term
% additions in the ANN structure.
m = size(X, 1);

% Apply nodal weights to input layer data and feeds them into ReLU
% activation function.
h1 = relu_sim([ones(m, 1) X] * Theta1');
% Apply nodal weights to hidden layer data and feeds them into softmax
% activation function.
h2 = softmax_sim([ones(m, 1) h1] * Theta2');

% Correction for number rounding.
%*************************************************************************
% Makes sure that there are no actual ones in the final h2 output matrix.
% Subtracts a very small number from found 1's in order to make them
% slightly less than one without overtly affecting the results of the
% neural network.
h2(h2==1) = h2(h2==1) - 1e-15;

% Calculation of Search Algorithm Position Fitness
```

```
%*************************************************************************
% Finds the highest output label probability of data set entries.
[¬, p] = max(h2, [], 2);
% Find the average percent rate of predictions matching ground truth.
ANN_acc = mean(double(p == y)) * 100;
% Create a confusion matrix for the tested ANN.
confuse_mat = confusionmat(y,p);

% Convert data set labels into bitwise format.
I = eye(10);
Y = zeros(m, 10);
for i=1:m
    Y(i, :)= I(y(i), :);
end

% Regularization term calculation
reg_theta1 = Theta1(:,2:size(Theta1,2)); % Remove bias terms from the
                                         % Theta1 weight matrix, as the
                                         % bias weights should not be
                                         % regularized.

reg_theta2 = Theta2(:,2:size(Theta2,2)); % Apply the same procedure to the
                                         % Theta2 weight matrix.
reg = (lambda/(2*m))*(sum(sum((reg_theta1.^2),2))...
    +sum(sum((reg_theta2.^2),2))); % Calculate L2 regularization term.

% Calculate the cost function based on outputs and converted data set
% labels. Applies previously calculated regularization term.
fit = (1/m)*sum(sum((-Y.*log(h2))-(1-Y).*log(1-h2),2))+reg;

end
```

B.4.2 Nodal Activation Functions

```
function [out] = relu_sim(in)
% Artificial Neural Network Rectified Linear Unit Activation Function
% Created by: Patrik Gilley
% Date Created: December 5, 2019
% Date Last Modified: December 5, 2019
% Program Name: relu_sim.m
%
% Program Inputs:
%   in = Forward pass input to the ANN layer that is using the ReLU
%        function as its activation function.
%
% Program Outputs:
%   out = Data outputs of the ANN layer that is using the ReLU function as
%         its activation function.
%
% This program implements the Rectified Linear Unit (ReLU) neural network
```

```
% activation function. This activation function is commonly used for hidden
% layer nodes due to its simplicity and the fact that it avoids the
% nodal weight saturation issues of a sigmoid function. The ReLU function
% preserves all positive values while zeroing out negative values.

% ReLU Activation Function Transformation of Input Data
%*************************************************************************
out = in.*(in > 0);
end
```

```
function [output] = softmax_sim(input)
% Softmax Activation Function
% Created By: Patrik Gilley
% Date Created: December 5, 2019
% Date Last Modified: December 5, 2019
% Program Name: softmax_sim.m
%
% Program Inputs:
%   input = The inputs to the ANN layer.
%
% Program Outputs:
%   output = The transformed outputs of the ANN layer.
%
% This program is an implementation of the softmax ANN activation function.
% The softmax function is traditionally used in multi-class classification
% artificial neural networks. Its advantage over the sigmoid function
% comes from the fact that it normalizes nodal output probabilities into
% the range of 0 to 100%. As such, it's used primarily as the activation
% function of a neural network's output layer.

% Softmax Equation
%*************************************************************************
output = exp(input)./sum(exp(input),2);
```

B.5 Utility Function Code

```
function save_my_figs(h,filepath,filename)
% Search Algorithm Figure Saving
% Modified by: Patrik Gilley
%   NOTE: This code was sourced from Dr. Yan. Small modifications have been
%   made to add an adaptive file saving path for better use with
%   Simulation_Frame.m's Monte Carlo simulation approach.
% Date Created: April 10, 2019
%
% This program is intended to provide a method of saving figures generated
% by the SA programs. It was bundled off into a separate function to ease
```

```
% the implementation of any modifications to file save types and/or number
% of files saved. This program uses a filepath specified by the user to
% save figures with a user-specified file name wherever they are needed.
%
% Program Inputs:
%   h = Figure being saved. This input should be passed to the function as
%       a handle of the current figure, as that is the format required to
%       properly save the figure data.
%   filepath = Directory/folder path to destination folder that the file
%              should be saved to. Must match the file path syntax of
%              platform that the program is being run on. Input should be
%              passed as a character string.
%   filename = Name that the saved file should use. Input should be passed
%              as a character string.

%printpdf(h,filename);
%savefig([filename,'.jpg'],'jpeg','-rgb','-crop', '-r250');
% Save the figure as a .pdf file.
saveas(h,[filepath '\' filename,'.pdf']);
% Save the figure as a .jpeg file.
saveas(h,[filepath '\' filename,'.jpg']);
% Save the figure as a MatLab .fig file.
saveas(h,[filepath '\' filename,'.fig']);
%saveas(h,[filename,'.eps'],'epsc');
```

```
function [temp_mid]=confusemat_plot(confuse_mat, sol_select, iter,...
    ann_fit, max_iter, iter_interval, inter_fig, will_pause)

% Confusion Matrix Plotter
% Created By: Patrik Gilley
% Date Created: May 22, 2019
% Date Last Modified: May 22, 2019
% Program Name: confusemat_plot.m
%
% Program Inputs:
%   confuse_mat = Confusion matrix data of specific solution being plotted.
%   sol_select = The spark/firework/particle that will be used to generate
%                the confusion matrices.
%   iter = The iteration count of the ANN training code.
%   ann_fit = The loop count of the fitness evaluation loop.
%   max_iter = The maximum number of iterations allowed in the overall
%              search algorithm program.
%   iter_interval = Sets the number of iterations the search algorithm
%                   must execute before creating an intermediate confusion
%                   matrix.
%   inter_fig = Passes a figure handle to the intermediate confusion matrix
%               plotting code to set where the confusion matrix plot goes.
%   will_pause = Controls whether the program pauses execution every time
%                an intermediate confusion matrix is generated, or pauses
%                every iter_interval iterations.
%
```

```
% Program Outputs:
%   temp_mid = Figure handle for the intermediate confusion matrix plot.
%              For use in later executions of the program.
%
% This program was created to package an iterative confusion matrix
% plotting mechanism into a function to ease how it's adapted across
% multiple search algorithms. It will plot confusion matrices for the ANN
% performance on a chosen solution found by a search algorithm, e.g. a
% specific particle for the PSO variants. Separate confusion matrices will
% be generated for initial and final solutions, and an intermediate
% confusion matrix will be plotted and updated/replotted every ten search
% algorithm iterations. An important note to make about the structure of
% this code is that it creates and outputs the handle of the intermediate
% confusion matrix figure that will be continually updated as temp_mid.
% This handle will need to be passed back into the program as inter_fig for
% this program to work as intended. Alternatively, other figure handles can
% be passed into this program through inter_fig if there is a need to have
% specific iterations on separate figures for later use. Those figures will
% need to be created before this program is run, as the code only creates
% a figure for the initial intermediate confusion matrix plot; the updating
% relies on referencing a figure that already exists.

% Confusion Matrix Generation
%*************************************************************************
if(ann_fit == sol_select)   % Allows for selection of specific
                            % particle/firework/spark for use with
                            % confusion matrix plotting, to limit the
                            % effects of the code on performance.

    % Initial Confusion Matrix Generation
    %*********************************************************************
    if(iter == 1)
        temp_mid = figure;
        confusionchart(confuse_mat);
        title('Initial Confusion Matrix (Iter = 1)')

    % Final Confusion Matrix Generation
    %*********************************************************************
    elseif(iter == (max_iter+1))
        temp_mid = figure;
        confusionchart(confuse_mat);
        title(['Final Confusion Matrix (Iter = ' num2str((max_iter+1)) ...
            ')'])

    % Intermediate Confusion Matrix Generation - Start
    %*********************************************************************
    elseif((iter-1) == iter_interval)
        temp_mid = figure;
        confusionchart(confuse_mat);
        title(['Intermediate Confusion Matrix (Iter = ' num2str(iter) ')'])
        if(will_pause)
            pause
        end

```

```
80      % Intermediate Confusion Matrix Generation - Continuation
81      %*************************************************************************
82      % Regenerates the confusion matrix every iter_interval iterations.
83      elseif(~mod((iter-1),iter_interval) && ((iter-1) ~= iter_interval))
84          temp_mid = figure(inter_fig);
85          confusionchart(confuse_mat);
86          title(['Intermediate Confusion Matrix (Iter = '...
87              num2str(iter) ')'])
88          if(will_pause)
89              pause
90          end
91      else
92          % Returns the given figure handle to ensure that the program always
93          % has an output.
94          temp_mid = inter_fig;
95      end
96  else
97      temp_mid = inter_fig;
98  end
```

```
1   % Principle Component Analysis Dimension Reduction for Artificial Neural
2   % Network Data Set Simplification
3   %
4   % Created by: Patrik Gilley
5   % Date Created: September 26, 2019
6   % Date Last Modified: September 26, 2019
7   % Program Name: proto_ann_pca_ana.m
8   %
9   % This program was written to reduce the dimensions of ANN training data.
10  % Specifically, this is meant to reduce the dimensions of the images of
11  % handwritten digits. A 28x28 image of a handwritten digit makes a feature
12  % vector of 784 features, which adds a lot of complexity in neural networks
13  % being trained to classify them. This program will run a Principle
14  % Component Analysis (PCA) of the image data set to create a plot that will
15  % identify the amount of information contained from a single dimension to
16  % the original amount of dimensions in the program. For example, a 28x28
17  % image has 784 features/dimensions, so this program will plot how much of
18  % the image information will be retained if it is compressed to one
19  % dimension, coompressed to two dimensions, and so on until the original
20  % amount of dimensions is reached which equates to no compression/dimension
21  % reduction being applied. The program will then compress the data set to
22  % the desired amount of dimensions.
23  %
24  % User-defined functions used:
25  %   featureNormalize.m: Used to normalize the features of the data set
26  %                       being used for PCA analysis/compression to an
27  %                       appropriate range.
28  %
29  % Note: [U,S,V] = svd(x) is used. U is a matrix containing eigenvectors,
30  % and S is a diagonal matrix containing eigenvalues.
31
```

```
% Clear Out Previous Program Executions
%*************************************************************************
clear;
close all;

% Load Target Data Set
%*************************************************************************
load('MNISTdata.mat');
X_train = train_images;
X_test = test_images;
X = vertcat(X_train,X_test);

% Normalize the features in the target data set.
[X_norm] = featureNormalize(X);

% Determine the number of samples and features in the target data set.
[m, n] = size(X_norm);

% Calculates the covariance matrix of the target data set.
sigma = (1/m)*(X_norm'*X_norm);

% Calculates the singular value decomp of sigma and returns the
% eigenvectors and eigenvalues.
[U, S] = svd(sigma);

% Controls whether the compressed data set generated by the program is
% saved. save_Xred = 1 enables results saving, and save_Xred = 0 disables
% it.
save_Xred = 0;

% Individual and Total Explained Variance Calculation and Plotting
%*************************************************************************
% Sum up all of the eigenvalues for determining the ratio of individual
% value variances.
tot_eigen = sum(S,'all');

% Determine the individual explained variance of each eigenvalue.
indiv_var = sum(S./tot_eigen);

% Turn the individual explained variance matrix into a cumulative vector
% that shows the amount of information captured by any number of compressed
% data set dimensions.
cumul_var = cumsum(indiv_var);

% Variance plotting
figure
plot(1:length(cumul_var),cumul_var)
title(['Number of PCA Components Used vs. Percentage of Target'...
    'Data Variance Captured'])
xlabel('Number of PCA Components')
ylabel('Cumulative Explained Variance')

% PCA Dimension Reduction of Target Data Set
%*************************************************************************
```

```
pca_comp = input('Enter the amount of PCA components to be used: ');

% Test to see if the number of PCA components that the data set will be
% compressed to result in a square picture, e.g. if pca_comp=225, resulting
% pictures will be 15x15.
pict_size_test = sqrt(pca_comp);
% Display a warning if the pictures are no longer square.
if(floor(pict_size_test) ~= pict_size_test)
   disp('Warning! Compressed pictures are not square!')
end
u_reduce = U(:,1:pca_comp);  % Captures the eigenvectors to be used.

% Initialize the matrix that will hold the compressed data set.
X_reduced = zeros(size(X, 1), pca_comp);

for ex_count = 1:size(X,1)
    X_reduced(ex_count,:) = X(ex_count,:)*u_reduce;
end
train_images = X_reduced(1:60000,:);
test_images = X_reduced(60001:70000,:);

% Saving of Compressed Data Set
%*************************************************************************
if(save_Xred)
    save('MNISTdata_pca.mat','train_images','test_images',...
        'train_labels','test_labels','U','pca_comp')
end
```

```
function [X_norm, mu, sigma] = featureNormalize(X)
%FEATURENORMALIZE Normalizes the features in X
%   FEATURENORMALIZE(X) returns a normalized version of X where
%   the mean value of each feature is 0 and the standard deviation
%   is 1.
%
%   Code sourced from Dr. Ng's machine learning course on Coursera.
mu = mean(X);
X_norm = bsxfun(@minus, X, mu);
sigma = std(X_norm);
% This if statement is to fix a problem with the standard deviation calcs
% for ex4data1.mat usage in the PCA code. Several columns have no standard
% deviation, causing undefined/Inf/NaN results that cannot be used in svd
% function.
if (ismember(0,sigma))
   sigma(find(sigma == 0)) = eps;
end
X_norm = bsxfun(@rdivide, X_norm, sigma);
end
```